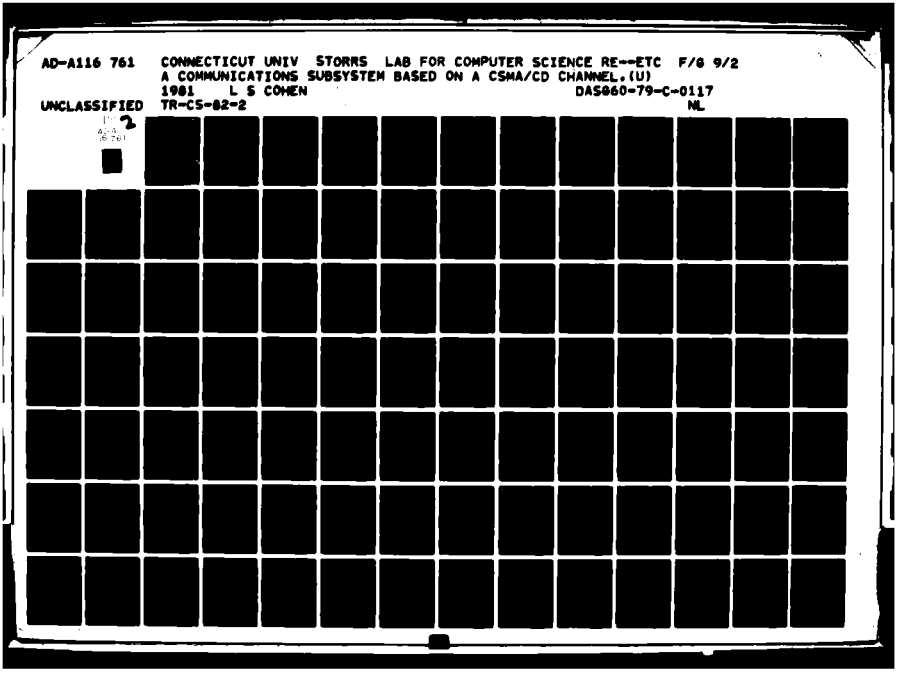
AD-A116 761 CONNECTICUT UNIV STORRS LAB FOR COMPUTER SCIENCE RE--ETC F/G 9/2
A COMMUNICATIONS SUBSYSTEM BASED ON A CSMA/CD CHANNEL.(U)
1981 L S COHEN DAS860-79-C-0117
UNCLASSIFIED TR-CS-82-2 NL



COMPUTER SCIENCE TECHNICAL REPORT

Laboratory for Computer Science Research
The University of Connecticut

COMPUTER SCIENCE DIVISION

Electrical Engineering and Computer Science Department
U-157
The University of Connecticut
Storrs, Connecticut 06268

A COMMUNICATIONS SUBSYSTEM BASED ON A CSMA/CD CHANNEL

Lawrence S. Cohen

Technical Report CS -82-2 ,

A COMMUNICATIONS SUBSYSTEM BASED ON A CSMA/CD CHANNEL

Lawrence S. Cohen

B.S., University of Connecticut

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

at

The University of Connecticut

1981

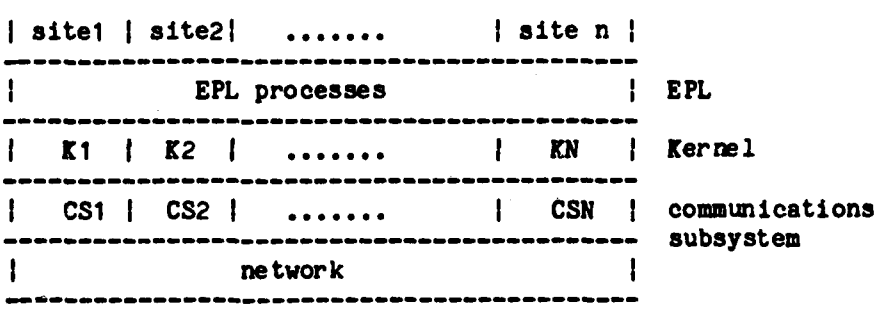
TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Current research [EEB80] at the University of Connecticut has produced an experimental distributed computing system (XDCS1) which is being used to study the structure, organization, and performance of software to control distributed systems. The computing system consists of four elements or layers as shown in the figure below:



The outermost layer is the applications language - in this case EPL [MAY]. Below this is the operating system kernel which implements the primitive functions of EPL as well as memory management. The communcations subsystem layer provides an interface between the kernel and the network hardware. The network, at the lowest level of the system,

consisted of five LSI-11 computers fully connected by serial links.

With the acquisition of a CSMA/CD network based on ETHERNET [DEC80] the need to create a new communications subsystem arose. Because the CSMA/CD network is used as a packet switched network (as opposed to the circuit switched, fully connected, serial link network of XDCS1) problems to be dealt with include: flow control of packets in the network, buffering of packets, and an appropriate communications protocol. Depending on the degree of message transport reliability required, the amount and complexity of the software could become enormous. Communications protocols based on Positive Acknowledgement Retransmission (PAR) schemes were considered as well as a technique which is based on the concept of virtual ciruit flow control [SUNSH81].

Thus the major objective of this thesis is to implement and document the design of an operating system which uses the CSMA/CD channel. A desired byproduct of the design is the creation of a family of XDCS-like systems. The parameters which differentiate family members are the interprocess communications model implemented by the kernel and the underlying network and associated communications subsystem software. The success with which the family has been created will be judged by the extent software is shared among family members and the ability to easily modify one member to obtain another. Finally the performance of the new system, based on the CSMA channel, must be evaluated and compared to the other family members.

The fundamental characteristics of the XDCS family are described

in chapter two. Much of this discussion is based on the work of J. Morse [JAM81]. Chapter three presents the design of the operating system which was being used at the beginning of this experiment (XDCS1). The reasons for this chapter are twofold. First it provides the only unified documentation on this version of the operating system. Another design is fully documented in [FONT80]. The second reason is to provide the reader with the background needed to understand subsequent design decisions.

Chapter 4 deals more specifically with design of a new communications subsystem. The CSMA/CD network will be described in some detail. The problems and solutions relating to implementing a communcations subsystem with this network are also discussed here.

Chapter five contains measurements which determine the relative performance of the CSMA/CD based system (XDCS2) and the serial link system (XDCS1). The effects of the new communcations subsystem software will be investigated as well as any performance gains related to the increased bandwidth of the CSMA/CD channel.

Chapter six is a summary of the work accomplished.

Chapter 2

Characteristics of the XDCS Family

2.1 Introduction

Since the XDCS system was already functioning at the time of this experiment, a design criteria was to use the existing software as much as possible and redesign only when necessary. As such this is not a formal attempt to create a new family, but the result should possess many of the characteristics which some researchers call a Family.

Much of the work in the area of families of software systems has been done by Parnas [PARNS72]. Habermann [HAB76] has applied these techniques to the development of operating systems in particular. In the following sections the design of an operating system, which closely resembles the XDCS family of computer systems is presented [MORSE80]. These sections will also provide an overview of the actual system upon which experimentation was done.

2.2 The XDCS Family

The goal of this project [EEB80] was to develop an operating system for multiple loosely-coupled mini-computers. The design of the system was to be independent of the type of CPU on which the operating system was run, and on the communications facility available to link the CPU's. Such a design should make it possible to implement a family of operating systems. The possible variations among family members included:

1. Implementation for several high level languages.

2. Incorporation of new operating systems features such as a new interprocess communications model (IPC).

3. Implementation for several different interconnection structures.

Figure 2.0 illustrates the XDCS family. Each path in the tree defines a family member. Each node represents the software or hardware which

is common to the family members.

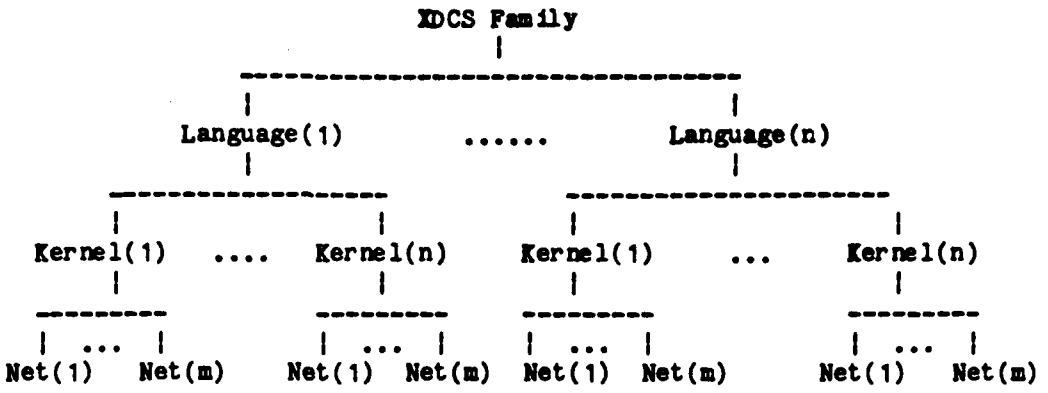


Figure 2.0 - The XDCS Family

The current operating system supports the programming language EPL [MAY79]. EPL is based on the principle of communicating sequential processes and is similar to CSP [HOARE78] and Distributed Processes [BRIN78]. Processes are autonomous and they communicate and synchronize with each other only through message passing. EPL requires the operating system to implement the following primitive operations:

-- SEND a message to a process

-- RECEIVE a message from a process

-- CREATE a process

-- RUN (initiate) a process

Two basic abstractions to be implemented by the operating system nucleus are processes and messages.

2.3 Requirements of the distributed system.

The requirements of the operating system can be summarized as follows:

1. The operating system kernel should support process and messages.

2. Processes can do the the following with repect to other processes:

   i. Create a process.

   ii. Run a process.

   iii. Send a message to another process.

   iv. Receive a message from any processes.

   v. Receive a message from a specific process.

3. A process can terminate itself.

4. After a process is created, there exists no "parent-child" relationship between the created process and the creating process.

5. Processes can be viewed as finite state machines. They can change state independently of any other process except while executing a SEND or RECEIVE.

6. The result of a RECEIVE depends solely on the text of the message transmitted by a SEND operation.

7. CREATE and RUN have no affect on the state (local variables) of the process executing them.

8. The site at which processes are located has no effect on the operation of the distributed system.

Some of these abstract requirements can be refined further. The requirement that processes make independent progress and may be regarded as independent state machines suggests that the concept of a process splits into 3 sub-concepts: 1) each process has a current state. 2) each process has a set of rules for making transitions from state to state (the kernel level protocol). and 3) there exists an operating system dispatcher/scheduler which causes processes to make state transitions.

The scheduler ensures fair treatment to all processes ready to receive service. More importantly it provides one of the fundamental goals of the operating system - to keep the CPU busy advancing the state of runable processes whenever possible. A process descriptor contains all state information associated with a particular process. A state transition diagram for the XDCS1 is given in appendix 2.

To incorporate processes located on remote CPUs, it is only necessary to provide a communications facility (subsystem) so that the remote processes can be accessed. All knowledge of the communications hardware and system topology are isolated to the communcations subsystem (CS). The CS will determine whether the destination of a message is local or remote. Eventually the message will be transmitted to the destination process on the appropriate site. The goal with this approach is to allow for changes in both the communications facilities and system topologies with no effect to the other parts of the operat-

ing system (i.e the operating system kernel) and to allow a new IPC with the same communications facilities.

2.4 Assumptions made in the XDCS family.

As stated earlier the distributed language should be similar to Hoare's CSP. The language used presently is EPL. Other languages which could be incorporated in the future include: Distributed Processes [BRIN78] or ADA [DOD80].

The kernel implements the primitive operations of the distributed programming language. The implementation of these primitive operations is determined by the interprocess communications model (IPC) assumed by the kernel. The present version of the kernel assumes that each site knows only about the state of processes created locally. Another assumption is that messages are delivered directly from the source process to the destination process without buffering. The latter assumption implies that a sending process cannot transmit a message to the receiving process until the receiving process is ready to accept that message. The kernel implements the interprocess communications protocols to obtain the state of the destination process. Another important assumption which is made by the kernel is that all messages are transmitted without error by the communications subsystem. Other IPC models may assume that information about all processes in the system is replicated at all sites or that messages are buffered. The IPC could also assume that the communications subsystem is unreliable and that the kernel should be prepared to handle corrupted

messages or failures in the system.

The communcations subsystem (CS) is at the lowest level of the XDCS system. The CS provides a virtual machine to the kernel by hiding the details of the network. The model of the CS assumes that:

1. the error rate of the underlying network hardware is very low. Therefore the CS will not be required to retransmit messages which were corrupted by the hardware or a noisy environment.

2. broadcasting is not required.

3. a protocol must be implemented to ensure that the receiver is prepared to accept a message.

4. a protocol must be implemented to handle contention for the network.

Different communications subsystems will vary only in the underlying hardware and the implementation of the above model.

Chapter 3

Implementation of an XDCS operating system

In the previous chapter the requirements of the family were defined. A design which fulfills these requirements is now presented.

The software for this implementation is devided into three levels. Figure 3.0 shows the general structure of the system. The arrows indicate the direction of information flow. Higher levels generally use functions provided by lower levels.

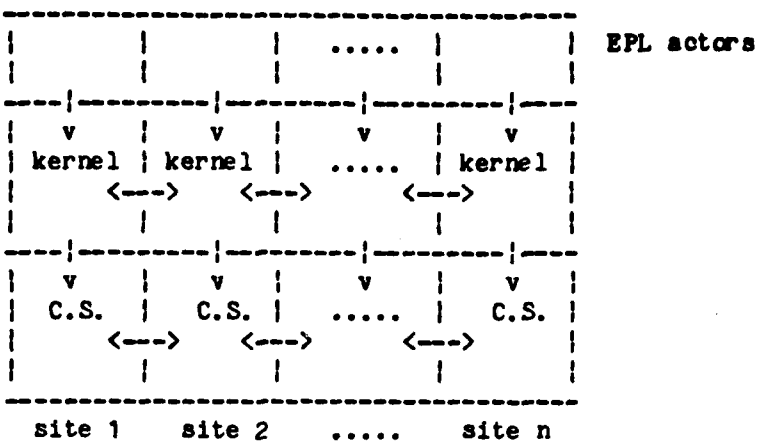


Figure 3.0 - XDCS System Structure

At the top level are the EPL processes. Beneath this level is the kernel. The kernel implements the basic EPL primitives: SEND, REC, RECF, CREATE, TERMINATE, and SYS. The kernel possesses no knowledge of the network except for the maximum number of sites in the system and its unique site number. At the lowest level is the communications subsystem (CS). When one kernel needs to communicate with another kernel, it submits a message to the CS. The CS will determine the destination site and transmit the message to the appropriate process.

3.1 EPL - Experimental Programming Language

The current specification of EPL [EEB80] consists of the following primitive operations:

1. CREATE - CREATE is used by a "parent" process to enter a new process ("child" process) into the system. An EPL process is sometimes referred to as an actor. The code for the process is called an act. The child is allocated memory space on a specified processor. This

memory takes the form of two data structures: a process descriptor and a data segment (or runtime stack). The details of these data structures are discussed later.

2. RUN - The RUN primitive is used to pass initialization arguments to the child process. The child process is then added to a list of active processes.

3. FINISH - FINISH allows a process to terminate itself at arbitrary points.

4. SEND - A SEND call is made by a process to send a text message to (or synchronize with) another process. A basic assumption made here is that messages are unbuffered from the process' point of view. This implies that the receiver must be prepared to accept the message. If the receiver is not ready the sending process must wait.

5. RECF - If a process needs information from a specific process it must execute the RECF command. If the specified process is not prepared to send then the receiver must wait.

6. REC - This command is the same as RECF except any sender is an acceptable source of information.

7. SYSTEM - SYSTEM is for miscellaneous operations. Currently these include: reading and writing to terminals, starting and stopping a real time clock, resetting measurement variables, accessing the site I.D. and finding out the current number of sites in the system.

All of the above operations must be implemented by the kernel. Hence

they are also referred to as kernel calls. Before EPL executes a kernel call information is placed in predetermined locations. The kernel functions know what information to expect and where to find it. Upon completion, the kernel will deposit results in locations known to EPL. Generally, information is passed in the general purpose registers (assuming LSI11 processors). In this way an interface is provided between EPL and the kernel.

The specific information which is passed is dependent on the kernel call. The description of the information layout and diagrams for each primitive are given in Appendix 1.

3.2 The Kernel

In order to implement the EPL primitives the kernel performs two *major functions:*

1. Resource management

and 2. Process management

Resource management entails the allocation of memory to newly created processes, the naming of newly created processes, and the maintanence of any queues or lists in the system. It also includes the scheduling of processes for fair CPU utilization. Process management involves maintaning all processes in a valid state.

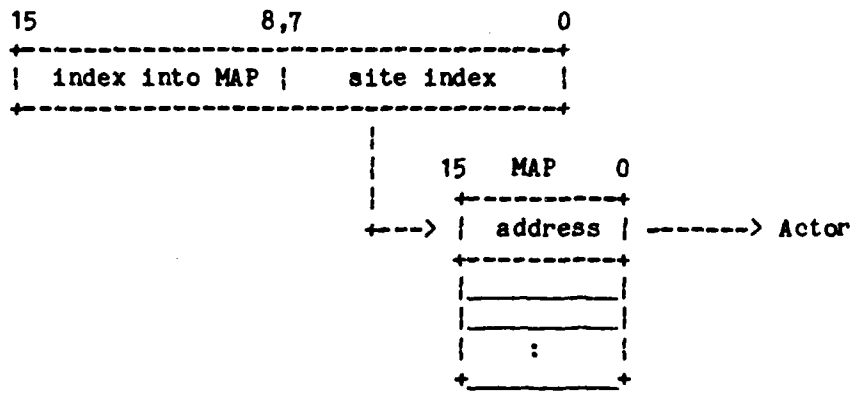
An important design decision lies in the distribution of system information [COHEN80]. Information concerning the state of processes has been partitioned so that each kernel knows only about local processes. A kernel must make enquiries to other kernels for informa-

tion about remote processes.

Since the kernel should possess no knowledge of a processes location, all messages are submitted to the CS regardless of whether they were local or non-local. This design decision helps reduce the complexity of the kernel by not treating local kernel calls as a special case.

3.2.1 Data Structures

Figure 3.1 illustrates all of the major data structures in the system. As stated in the previous chapter, when a process is created it is allocated memory in the form of two data structures: a process descriptor and its data segment. The process is also given a name which is unique in the system. The name is composed of two bytes of information:



The low order byte contains the CPU index where the process was created. The high order byte contains an index into an array called MAP. At this location in MAP is placed the base address of the process descriptor. MAP contains the location of every process descriptor local to its site. The kernel can now use these names to access infor-

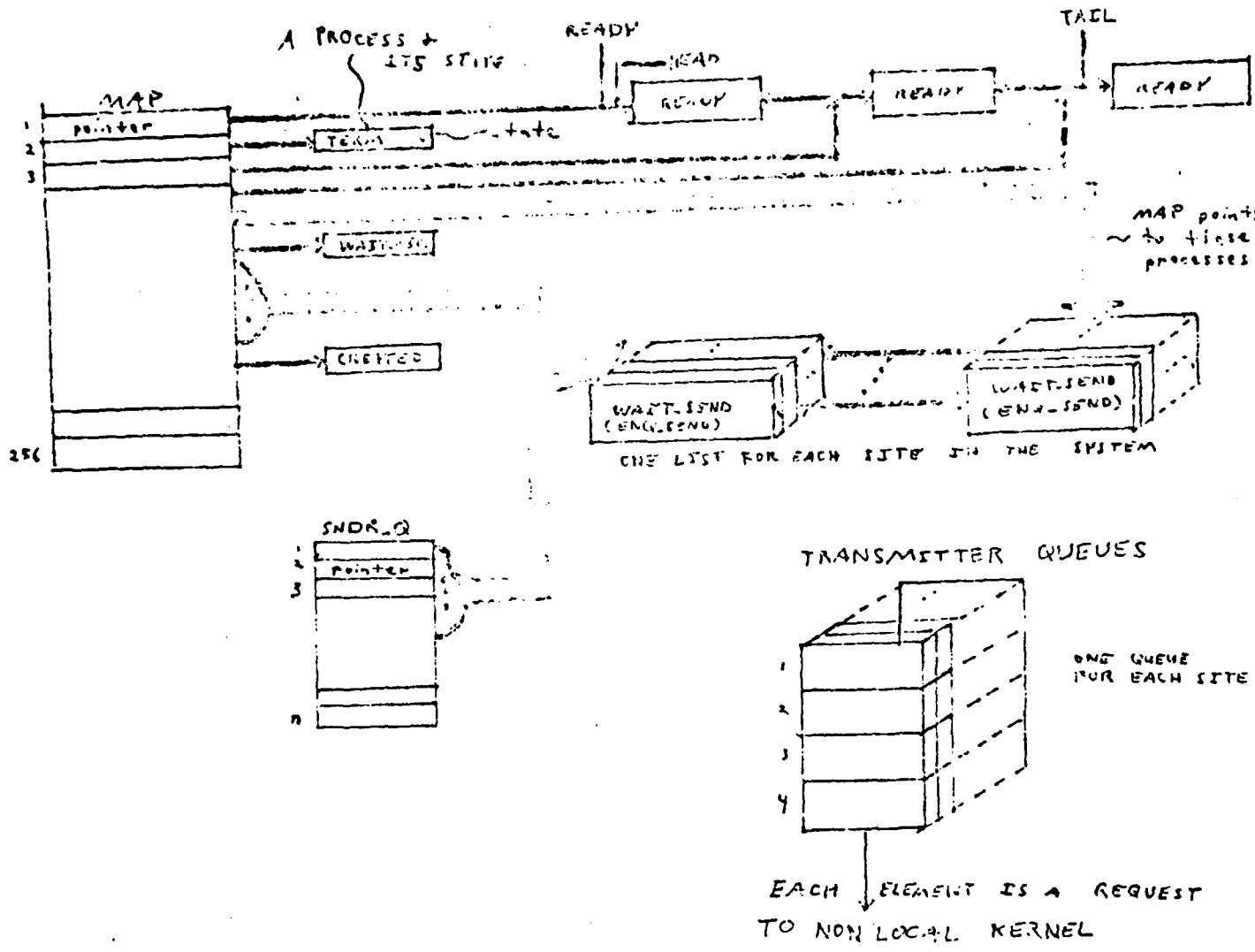


FIGURE 3.1 - XDCS Data Structures

mation about any process in the system.

Appendix 3 contains a detailed layout of a process descriptor. The size of the process descriptor can vary with the number of sites permitted in the system (MAX_SITES). The first word is used to indicate the success of the kernel call. Another slot holds the current state of the process. A "pending sender" field exists for each site in the system. These slots hold the number of messages which the corresponding site would like to send to this process. Two slots exist for linking process descriptors into lists. The last field contains the name of the process to which that process descriptor belongs. The data segment of a process is described in detail in appendix 1.

The kernel must maintain several queues. The local set of processes, ready to run on the CPU, is contained in a FIFO queue called the ready list. The variables READY.HEAD and READY.TAIL point to the head and tail of the list respectively. Processes which become ready to run are placed on the tail of the list. For the current implementation, when the CPU becomes available the next process to use it will be taken off the head of the list.

When a process executes a SEND command the kernel must determine whether the destination process is prepared to receive the message. If the receiver is not ready the sending process is placed on a queue of processes which need to transmit messages to that site. A queue exists for each site in the system. A list of pointers (SNDR_Q) con-

tains the head and tail of each queue of senders - Figure 3.1.

At the receiving site, the arrival of an enquiry is marked by incrementing two fields in the receivers process descriptor: total_pending and pending_sender[source]. Total_pending is the total number of processes at all sites which have requested to send a message to this process. Pending_sender[source] holds the number of processes which have attempted to send a message from a specific site.

Both the ready list and the sender queue use the two link fields in the process descriptor to create the respective lists. This is made possible be a design invariant which permits a process to be on only one queue at a time.

In order to transport a message the CS must be given enough information by the kernel to do so. The information needed is the name of the source process and destination process, the type of message to be transmitted, and the address of any parameters to be transmitted. The kernel will place the information in a queue (figure 3.4) which corresponds to the destination site. Each element in the queue represents a request to transmit a message to a remote site.

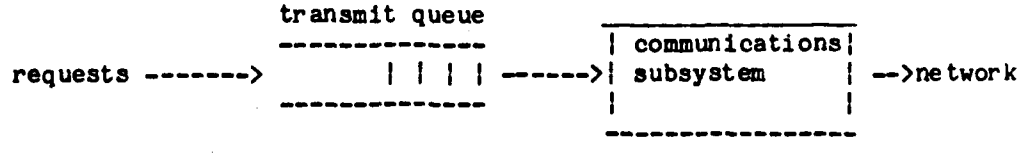


Figure 3.4 - Transmitter queue

Since the information provided to the CS includes process names, the CS must possess knowledge of the kernel data structures. The number of bytes to be transmitted is a function of the message type. The format for each type of message is given in Appendix 4.

When a message is received by the communications subsystem the inverse procedure occurs. Information needed by the kernel is placed in a variable - CMD. CMD has the same structure as one slot in the transmitter queues:

source	destination	function	parameters

The kernel function which corresponds to that message will now use CMD to complete or continue the kernel call protocol.

3.2.2 Summary of Kernel Call specifications

This section describes the actions each kernel call has to perform in order to complete the EPL primitive. The implementation of each call is dependent on the kernel level protocol.

1. INIT - INIT is called by a system start up routine. In this version of the kernel INIT is responsible for initializing all of the system data structures. It also initializes any I/O devices in the system. If the site index is zero, the first EPL process is created and started running.

2. CREATE - A create call is made by a parent process to enter a new process into the system. If the EPL program did not specify the site of creation, the kernel will choose the site. The current algorithm for choosing the site is :

site of creation = (local site index + 1) modulo (total number of sites)

A create request is made to the chosen site which includes the address of the ACT (code which corresponds to the child process). At the des-

tination site, the kernel will allocate memory for the new process, assign it a unique name, and then return the name to the parent process.

3. RUN - The RUN primitive is called by the parent process to initialize the child process. A message is passed to the child process which includes the number of initialization parameters to be sent followed by the parameters themselves. The child process will then be placed on its local ready list.

4. TERMINATE - TERMINATE is called by a process upon completion of its code. Its state is changed to indicate its terminated status.

5. SEND - A SEND call is made by a process that needs to transmit a message to another process. The "sender's" kernel will transmit an enquiry to the "receiver's" kernel. The receivers kernel will first mark the presence of the enquiry. The kernel will then determine the status of the receiver process. If the process is not prepared to receive, a negative acknowledgement is returned to the senders kernel. At the senders site the process which initiated the SEND will be placed on a waiting queue corresponding to the receivers site. If the receiver is ready, a positive acknowledgement is returned to the senders kernel. The message is then transmitted to the receiver.

This implementation of the SEND command does not allow the sending process to execute until the receiver process has accepted the message. This implies that the SEND can be used for synchronization as well as message transmission.

6. RECEIVE - The RECEIVE command is executed by a process which needs information from another process in order to continue. The receivers kernel will first check for enquiries from sending processes (pending senders). If there are pending senders, the kernel will send a message which will inform the sending kernel that the receiving process is prepared to receive a text message. At the senders site, the kernel will determine if there exists a process which wishes to transmit a message to the receiving process. The message will be transmitted if there is a sending process. Otherwise a negative acknowledgement will be returned. In the event of a negative acknowledgement, the receiving process will be placed in a waiting state.

7. SYSTEM - SYSTEM is described in appendix 1.

3.2.3 Process Management

A process may be viewed as finite state machine - Figure 3.5. The next state transitions are a function of the EPL primitive being executed by the source process, the state of the destinations process, and the current state of the source process.

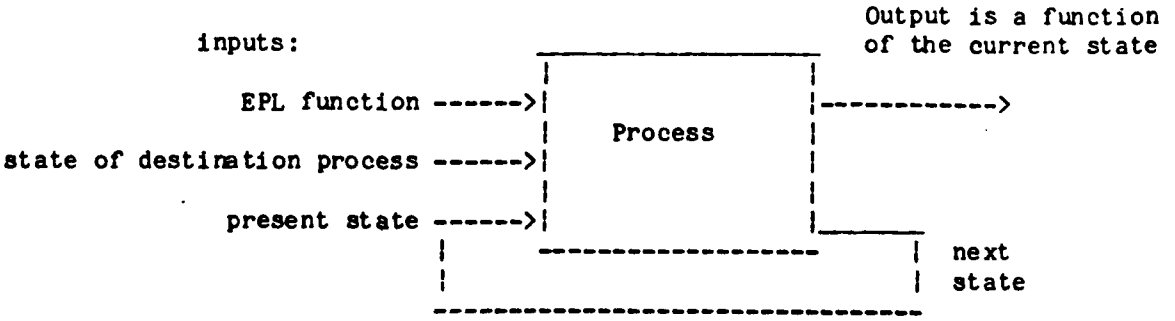


Figure 3.5 - Process as a finite state machine

The algorithm which governs the state transitions is defined by the kernel level protocol. The kernel includes the of routines which implement this protocol.

3.2.3.1 Process States

A state exists for each process which corresponds to a step in the execution of a kernel call. The possible states and their definitions for this kernel are as follows:

Process State Definitions

1. CREATED: the process has been created (i.e. it has a name), but no parameters have been transmitted by the creating process.

2. CREATING: the process is creating a child process and has not yet received the name of the child process.

3. TRANSMITTING: the process is either sending a message to a receiving process or it transmitting parameters to a child process.

4. READY: the process has finished a kernel call and is ready to execute but may not be assigned a CPU.

5. TERMINATED: the process has terminated.

6. ENQ_REC: the process is ready to receive a message from any sender. An enquiry is sent to the site of a pending sender. will either be negative or the expected message.

7. ENQ_RECF: the process is ready to receive a message from a

specific sender. An enquiry is sent to the site of that sender (only if there are pending senders for this process). The acknowledgement will either be negative or the expected message.

8. WAIT_REC: the process is waiting to receive a message and a suitable sender process has not yet enquired if the process is prepared to receive.

9. WAIT_RECF: the process is prepared to receive a message from a specific sender but the sender is not yet prepared to transmit.

10. ACK_ENQ: the process has received an enquiry from a potential sender and has returned positive acknowledgement to that sender.

11. ENQ_SEND: the process has sent a message to enquire if the receiver process is prepared to accept a message.

12. WAIT_SEND: the process is prepared to send a message but the receiver is not prepared accept it.

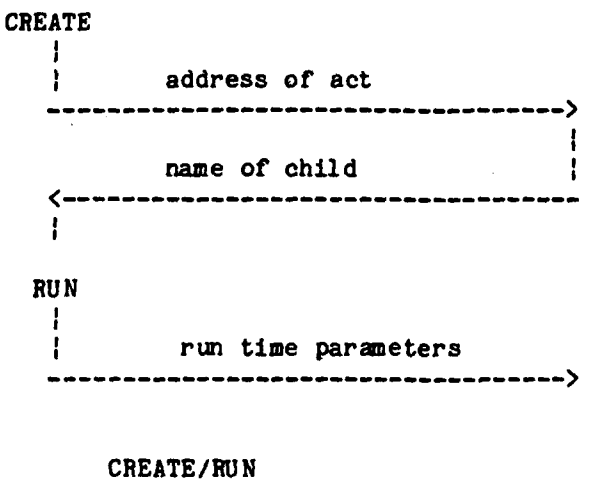
13. CONCURRENT: a process has sent a message to enquire if the receiver process is prepared to accept a message. The receiver has concurrently sent a message to the sender process to enquire if it is prepared to transmit it's message.

A state transition diagram can be found in appendix 2. The specific kernel routines which cause these transitions are identified and explained.

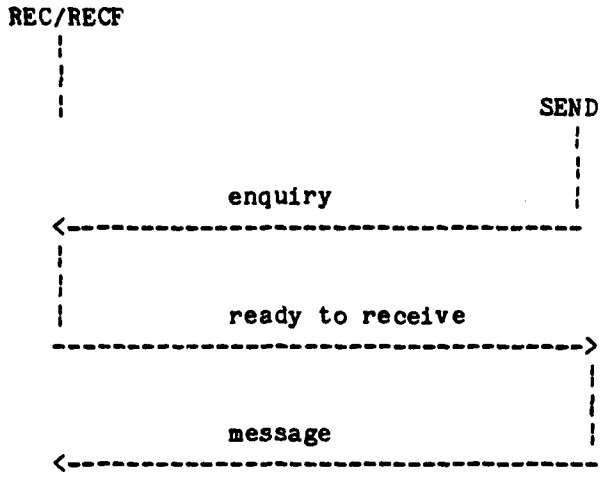
3.2.3.2 Kernel Level Protocols

While executing an EPL primitive the kernel may require information about processes on another or the same site. The protocol defines the correct sequence of actions the cooperating kernels must take in order to effect the information transfer. Each state of a process corresponds to a different point in the protocol.

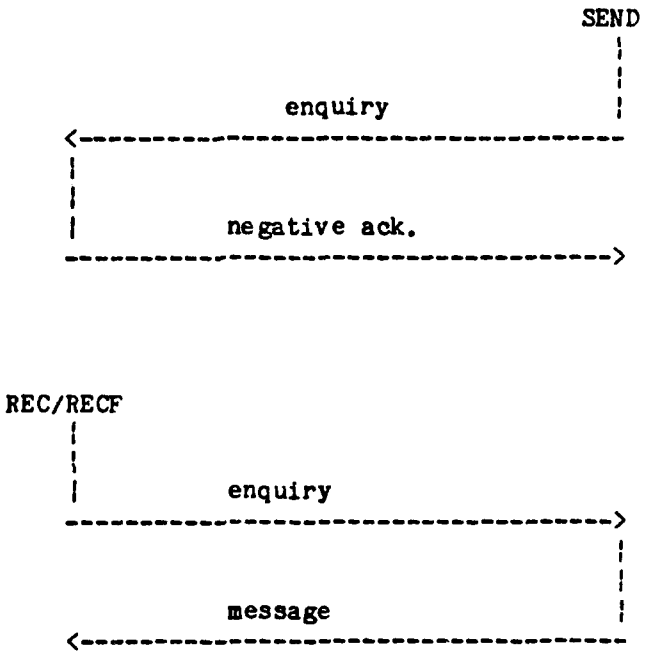
The kernel protocol is dependent on the EPL primitive. The following timing diagrams illustrate each protocol. The format of each message of the protocol can be found in appendix 4.



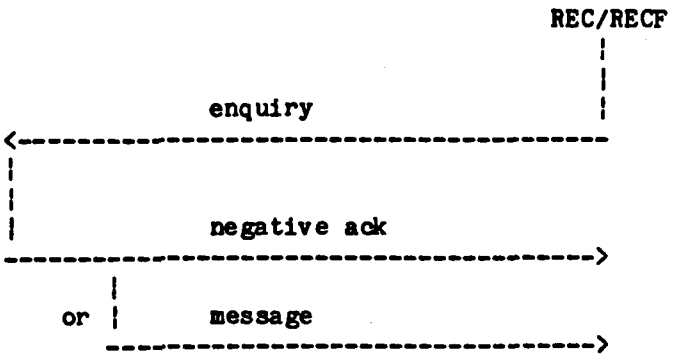
CREATE/RUN



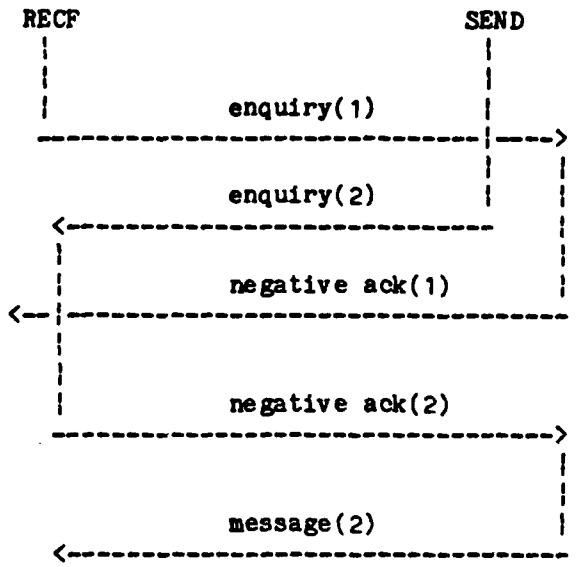
REC first with no pending senders



SEND first



REC first and pending senders



SEND and RECF concurrently with pending senders

3.2.4 Communications Subsystem for XDCS1

One of the criteria of the design of the communications subsystem (CS) is to "hide" the details of the network from the kernel. When the kernel needs to transport a message to another or the same kernel, it formats a request to the CS, places the request on a transmitter queue, and then informs the CS that a request has been made. The CS will determine if the message is local or remote. If local, the message is transferred to the local process. If remote, the message must be transmitted through the network using a well defined protocol.

The CS can be viewed as having three levels (Figure 3.6). At the top (TRANSMIT) is software for determining the destination (local site or remote site). This level will also ensure that a message will not be transmitted until a previous transmission has completed. Below this (NETWORK) are the details of the link level protocol. At the lowest level are the routines which interface with the hardware. The interface routines (INPUT and OUTPUT) are responsible for transmission and reception of all messages.

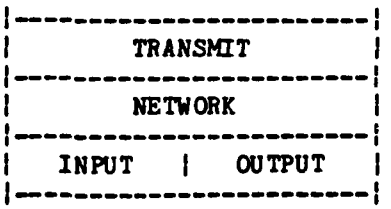


Figure 3.6. Structure of the Communications Subsystem

3.2.4.1 The Kernel/CS interface

As stated in the previous section a goal of the CS is to hide details of the network. At the same time it is desirable to hide details of the kernel from the CS. In XDCS1 the kernel places information needed by the CS in a transmitter queue. The CS can extract the destination site and the location in memory of any parameters to be transmitted. Implicit in this approach is that the CS possesses knowledge about the structure of process descriptors and data segments.

3.2.5 Link Level Protocol

The link level protocol is highly dependent on the network hardware [COHEN80]. The protocol in this case assumes a set of computers fully interconnected by serial links.

The protocol is based on the concept of connection and is very similar to the protocol used in another version [FONT80]. Figure 3.7 is a timing diagram of the protocol.

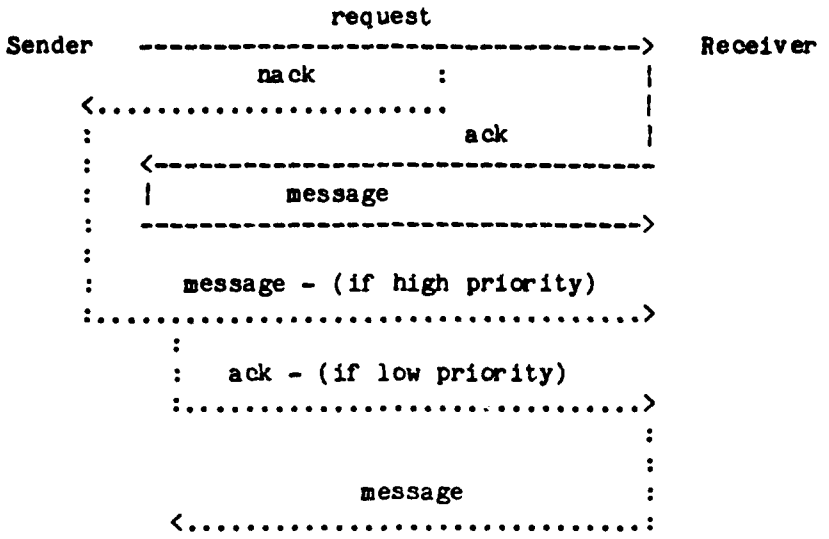


Figure 3.7 - Link Level Protocol

A request for a connection is passed from the a source CS to a destination CS. The CS initiating the communications then waits until the CS sends an acknowledgement. Once a connection is made, processors are dedicated to the information exchange and remain connected until all the information has passed.

Because the basic protocol requires the initiating CS to wait for and acknowlegement from the cooperating CS, it is possible for deadlocks to occur. Another problem is that it is possible for each of two I/O subsystems to simultaneously attempt to initiate a connection with the other. To prevent deadlocks and arbitrate contention, the nodes are arranged in a hierarchy according to their ID numbers. Lower numbers have higher priority. As with any hierarchical scheme,

there is a danger of starving the low priority processors.

The basic connection protocol has been extended to work with the hierarchy. A request for a connection is passed from the CS initiating the communications to the CS it wishes to communicate with. The initiating CS then waits for an acknowledgement before proceeding. If it is a positive acknowledgement the message is sent. If it is a negative acknowledgement, indicating that the cooperating CS is trying to initiate communications on the same link, its handling depends on the relative positions of the two sites. The CS with lower priority relinquishes the line, sends a positive acknowledgement and then waits to receive the message. After the message has been transmitted the CS with higher priority will send a positive acknowlegement to the lower priority site and then prepare itself to receive the message. The lower priority CS can then transmit the message.

While a CS is attempting to make a connection without priority, in order to prevent deadlocking, it must listen to kernels above it in the hierarchy. If one of these wishes to make a connection, the low priority CS sends a positive acknowlegement and receives the message before returning to its own communication. Once it has made the connection, it no longer listens to the other communications subsystems.

Chapter 4

Communications Subsystem Design for XDCS2

As was stated in the introduction to this thesis: the primary objective was to design a new communications subsystem based on a CSMA/CD network. Before a specific design is undertaken the general structure of communications subsystems will be examined.

The model for a communications subsystem for the XDCS system is discussed in Chapter two. Implementations of this model can vary with respect to the services provided by the communications subsystem and the underlying network.

Recent efforts, by the ISO to standardize the decomposition of abstract network functions [ISO79], list the services which should be provided by CS software. A summary of their results will be the first topic of this chapter.

It will be shown that the services needed for a serial link network are slightly different than for a CSMA/CD network. After the

services required by the CSMA/CD are determined, a specific implementation will be described.

4.1 Abstract Communications functions - ISO Model

Since a distributed computing system may be viewed as a local area network some of the techniques used to communicate between nodes in a network may be used here. In particular, the ISO has attempted to use layering to define a communications subsystem architecture [ZIM80].

The proposed layers are shown in figure 4.0. Each layer represents a protocol or format which is needed to perform the function of that level.

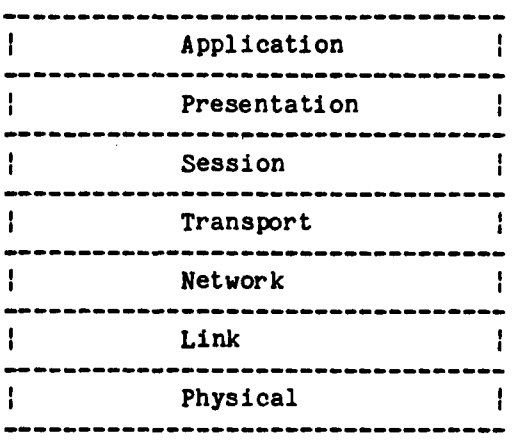


Figure 4.0 - Open System Architecture

At the lowest level are the physical/electrical standards. For example a distributed processing system may require each node to use the same kind of serial link interface. The specification of all con-

nector leads, voltages, etc are also provided at this level.

The link level is responsible for moving raw data over the network reliably. The link level may have some notion of the data format but it will use this knowledge for low level functions only - e.g. error detection and correction. This should be the only software in the system that interacts directly with the network hardware.

The network level is responsible for message transfer between nodes in the network. This level could also be referred to as the switching level. It establishes connections, real or "virtual", between two or more sites. Routing functions are also provided here.

The transport layer, in combination with the lower levels, should comprise a complete communications subsystem. Services provided here could include: flow control, end to end error control, sequencing of messages, and duplicate message detection. Provisions for virtual circuits should also be given here.

The remaining three levels: Applications, Presentation, and Session, provide for communication at the process level. In the XDCS family the EPL and kernel layers correspond to these levels. These higher levels should possess no knowledge of the system topology. It should be noted that the ISO standard only provides a guideline for decomposing network functions. Other networks have been constructed using the layering concept which do not fit precisely into this model. Examples include IBM's SNA [GRAY79],and Digital Equipment Corporations DNA [WECKER80]. Another example is the communications subystem for

XDCS1.

4.2 Abstract functions required by a serial link

The implementation of a communications subsystem based on serial links (XDCS1), is presented in chapter 3. This section now describes the relationship of XDCS1 to the ISO model of communications subsystem architecture.

It was stated in chapter 3 that the CS for a serial link network used only one protocol: the "link level" protocol. In terms of the ISO model this link protocol provides the functions of two ISO levels: the link level and the network level. The only link level service provided is raw data transmission. No error detection or correction is provided. The network level establishes an hierarchy among sites in the system and makes connections between the sites. No message routing is performed.

There is no logical equivalent of a transport layer in this system. This is not a problem because there is no need for the functions provided by this level. Flow control is effectively accomplished by the connection protocol used in the network level.

4.3 The CSMA/CD network.

The network used in XDCS2 is an example of a shared bus which uses a carrier sense multiple access with collision detect (CSMA/CD) communcations protocol [TOBAG79], [SWARTZ77]. The hardware used by the network consists of a coaxial cable and a set of Network Interface (NI) boards [DEC80]. Each site in the system requires the use of one

NI board to be part of the system - Figure 4.1.

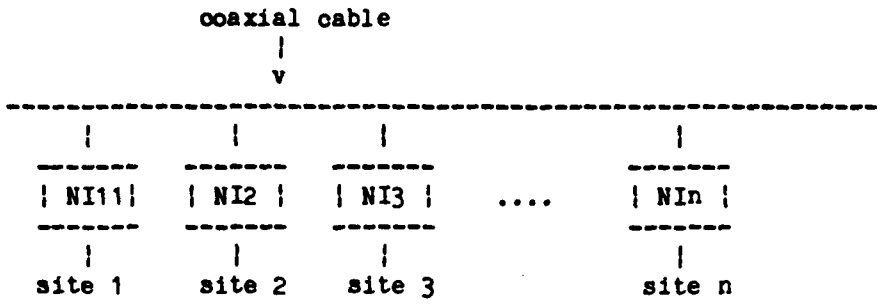


Figure 4.1. NI based network.

The properties of the network are very similar to Ethernet-like systems [METCALF76]. Each station in the network can communicate by serially transmitting packets of information over the cable to any other station which has tapped the cable. The packet contains destination address(s) and is copied from the cable by that destination(s). The destination site must specify the address of a buffer into which messages can be placed.

Control and access to the cable is completely distributed among all stations. Each NI board provides the facilities to implement the CSMA/CD link level protocol. Transmissions initiated by a site defer to any which may already be in progress. Once started, if interference with other packets (collision) is detected a transmission is aborted. After a collision has been detected the transmitting NI will "jam" the cable thereby ensuring that all sites involved in the transmission will detect the collision. The transmitting site can then retransmit the message if it so desires.

Some important characteristics and specifications which may con-

cern users of the NI boards include:

1. After detecting that the cable is idle the NI board will wait a prespecified time period before transmitting a message. The current design waits 256 microseconds.

2. The user must provide the address of a buffer into which a received message will be placed.

3. After the message is received a new buffer address must be provided before the next message arrives. Otherwise the buffer may be overwritten.

4. The NI must be "turned on" before each reception. Otherwise a message will be missed.

More detailed user information can be found in [MORSE80].

4.4 Abstract functions required by a CSMA/CD network.

In the serial link network described previously there exists a dedicated link (or circuit) between every site in the system. A message is transmitted to a specific site by switching to the link dedicated to that site. This is a circuit switched network.

All sites in an NI based network must share one bus. Messages or packets are placed on the bus. Each message contains its destination address. The destination site will receive any packet which is addressed to it. This is a packet or message switched network.

Communications on a NI (packet switched) network poses a different set of problems than the serial link (circuit switched) case.

Since there is no centralized control on the NI bus, two or more sites may try to transmit simultaneously. This will result in collision and destruction of packets.

Regardless of the network some procedure must be used to guarantee that the a site will be ready to receive all messages transmitted to it. These procedures can be categorized as flow control procedures. In the serial link network a connection had to be established before a message could be transmitted. Only one message could be transmitted per connection and only one site could establish a connection at a time. Attempts to make a connection by other sites are recognized by the destination site and serviced eventually. Hence there was a flow control limit of one message to a site.

If a node in the NI network is servicing a message it will not recognize any other message which might be destined for this site. There are two basic solutions to preventing these "missed" messages. One method entails acknowledging each or a group of messages from a site. The other ensures that the transmitter will wait until the receiver is ready. Both techniques have distinct advantages and disadvantages which will be discussed in the next section.

In any event a different set of services must be provided in the communications subsystem layers. The link level must be able to detect collisions and provide this information to higher levels. The network level no longer has to establish a connection between sites. Messages are merely submitted to link level procedures for transmission. Handling of collisions occurs at the network level. The tran-

sport level will be responsible for guaranteeing message reception. The techniques used to provide guaranteed transport are discussed in the following section.

4.5 Implementation of abstract functions for a CSMA/CD

The previous section described the functions required by the CSMA/CD network. At the link level two functions are required: collision detection and raw data transport. Collision detection is provided by the network interfaces. Collision handling protocols are provided at the network level. The current implementation of collision handling entails resubmitting the message to the link level if a collision is detected. Reliable message delivery is provided at the network level. As will be shown the design of software to provide reliable delivery of messages is influenced by the degree of reliability required by the system.

Reliability in this case is limited by the message error rate of the communications subsystem. Since the model of the communications subsystem assumes that the hardware is reliable, the error rate of the communications subsystem will be proportional to the error rate of the hardware. Because message errors are not detectable by the kernel, any such errors will cause the system to fail. If the system fails for the latter reason the EPL programs will have to be restarted.

The previous discussion did not mention the possibility of a receiver "missing " an uncorrupted message. Several basic techniques were considered to solve the problem and are discussed subsequently.

4.5.1 Positive Acknowledgement Retransmission (PAR) schemes.

Figure 4.3 is a timing diagram of a basic PAR protocol. Most PAR protocols are a variation on this theme [SUNSHINE81].

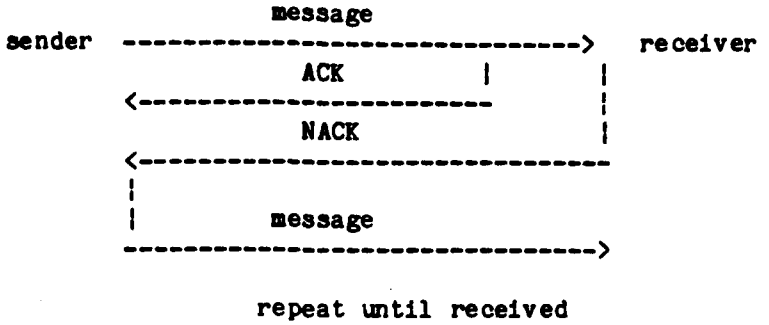


Figure 4.3 - PAR protocol.

A message is transmitted to a receiver. If a packet is received correctly a positive acknowledgement (ACK) is returned to the sender. This ACK merely implies that the transport layer of the receiving site has accepted the message. It does not imply that the destination process has received the message. If the packet was received in error, the receiving site may optionally send a negative acknowledgement (NACK). Since it is equally likely for a NACK, ACK or a packet to be lost or corrupted, this scheme in itself is not completely effective. The sending protocol must also use timeouts. After each packet is transmitted a timer is started and if an ACK is not received before a timeout period has elapsed, the data packet is retransmitted. If a NACK is received before the timeout the message can be retransmitted sooner than the timeout period. If an ACK has not been received after several retransmissions the sending site can take some special action,

such as notifying the user process.

Another situation which must be handled by the PAR protocol is a lost ACK. If an ACK is lost, the sending site will retransmit the packet. This will appear as a duplicate message at the receivers site. The receivers protocol must be able to differentiate between new messages and duplicate messages. A solution entails a unique identifier attached to the message by the sender. The receiver must keep track of the identifiers of packets successfully received. The receiver must also check all new packet I.D.s. If the packet I.D. is new, the I.D. is stored and an acknowledgment (ACK) is returned. If the I.D. is a duplicate an ACK is returned and the packet is discarded. Sequence numbers are commonly used as identifiers. The sender could sequentially number each message. The sequence number maintained by the receiver for duplicate detection may be considered the left edge of a "window" of acceptable sequence numbers. The size of the window is the number of packets the sender may transmit without having the first message acknowledged. Any message retransmitted in the window will be detectable as a duplicate.

4.5.2 Buffered Message Approach

A goal of the XDCS2 communications subystem is to provide the same degree of reliability that existed with the serial link version (XDCS1). To do this, a mechanism must be provided for preventing missed messages. PAR protocols present one solution to the problem. Another solution which takes advantage of the timing characteristics of the NI interface, was eventually adopted. A specification of the

NI ensures that after a message has been transmitted each node in the system must defer transmitting another message for a prespecified time period (T). The receiver has T seconds to accept the message and prepare itself for the next possible reception.

In XDCS1 messages were not buffered at the receiving site. The network protocol would transmit the message in two pieces. First a message header was transmitted which the reciever could use to determine the destination process. If there was more text, the sending site would extract the data from the source process and transmit it directly to the destination process. This scheme is not possible in the NI based network for several reasons.

First, a message must be transmitted as a complete unit. The NI interface is essentially a DMA device which transmits contiguous blocks of data. This implies that the entire message must be formed in one contiguous buffer before it can be transmitted. Since the receiver cannot predict which site will send the next message it cannot provide the address of the destination process to the NI. Therefore a prespecified buffer must be provided. After the message arrived the buffer could be analyzed, the destination process determined and the message transferred to that process. The problem here is that the interarrival time of messages (T) is very small (approximately 200 microseconds). It is not possible to transfer the recieved message to the destination process and prepare the NI interface for the next message in 200 microseconds. The only viable solution, then, was to buffer messages at the receiving site and perform the transfer to the

destination process at a later time.

A pool of buffers would be provided at each site - Figure 4.4.

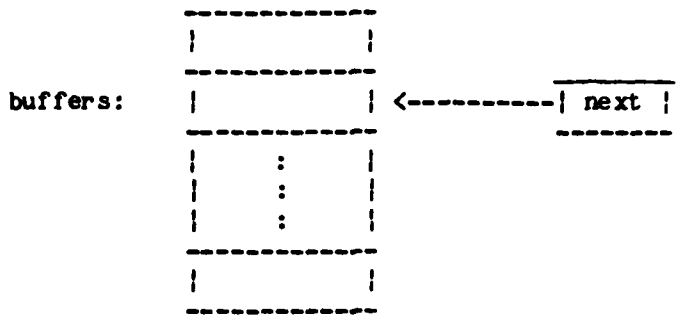


Figure 4.4 - Buffer pool for flow control procedures

The receiving protocol will merely make the address of the next available buffer known to the interface hardware. The function of transferring the data from the buffers to the processes will have to be provided by higher levels of software.

Given that a receiving site could switch buffers in T seconds, only one problem remained: limited buffer space. Due to memory constraints only a finite number of buffers could be provided for message reception. A mechanism was needed to prevent more messages to be sent then there are buffers available. In otherwords a flow.control algorithm was required.

4.5.2.1 Flow Control - an overview.

Flow control is a set of procedures whose purpose is to limit the flow of message traffic in a network [GERL81], [LAM75].

The technique developed in this section is based loosely on the concept of virtual circuit flow control [GERL81], [KLRK80]. A virtual

circuit is a logical connection between two communicating entities. Before a message can be transmitted the connection must be "opened". When the entities no longer need to communicate the connection must be "closed". Once the connection has been established the sender is assured that the receiver is prepared to accept a prespecified number of messages. After this prespecified number of messages has been sent, the sender must wait until the receiver has indicated that it is ready to accept more messages.

To implement this kind of flow control in an XDCS system a mechanism must be provided for establishing a circuit and providing the sender with an acknowledgement that a buffer is available. A design criteria which also should be met is that any change to the operating system to implement the flow control algorithms will work with both the serial link network and the NI network. The serial link version can be used as a test of the flow control layer which uses a "working" communications subystem. The flow control will be viewed as a new layer in the XDCS family. Flow control layer will be placed

above the communications subsystem and below the kernel - Figure 4.5.

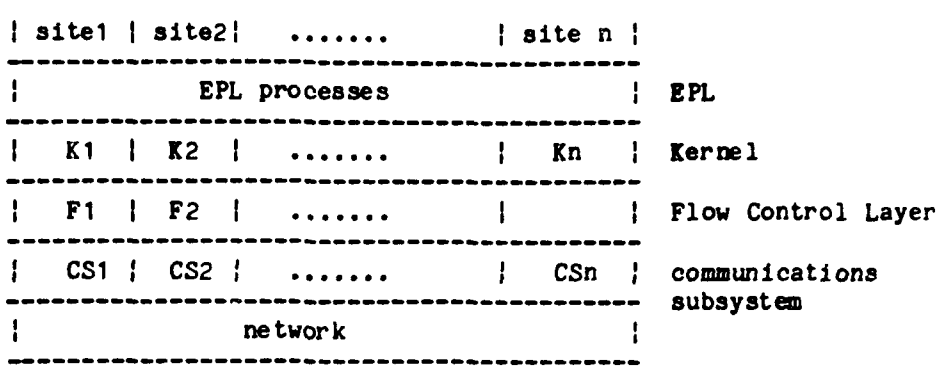


Figure 4.5 - XDCS system with flow control

Three family members will now exist:

1. A serial link network without flow control - XDCS1.

2. A serial link network with flow control - XDCS3.

3. The NI based network - XDCS2.

Any message which must be transmitted will now be submitted to the flow control layer. The flow control layer will determine if a buffer is available at the receiving site. When a buffer becomes available the message will be submitted to the communications subsystem and subsequently transmitted. The communications subsystem at the receiving site will be responsible for placing the message in an available buffer.

4.5.2.2 Flow Control implementation - Virtual Circuits.

The flow control layer makes use of the fact that kernel level protocols are half duplex operations. In otherwords, the kernels involved in a kernel call will serially transmit messages to each other until the protocol is complete (Chapter 3). For each protocol in progress only one buffer is needed at each site: a transmit buffer and a receive buffer. After a message of the protocol has been transmitted the transmit buffer can be used to receive a reply - figure 4.6.

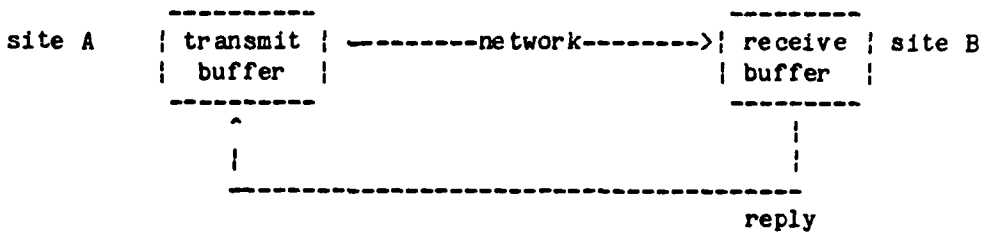


Figure 4.6 - Virtual Circuit buffers

A virtual circuit, then, consists of a pair of buffers: one at the kernel initiating the protocol and one at the destination kernel. Before a protocol can begin, the flow control layer will check to see if a virtual circuit is available between the opposing sites. If none is available the process which initiated the kernel call will be placed on a waiting list until a virtual circuit becomes available. After a virtual circuit has been granted to a process, the protocol can proceed. The virtual circuit belongs to the initiating process until the end of the protocol. The destination kernel will use the same circuit to send replies.

The maximum number of messages a site can send to another site -

the "flow limit" - corresponds directly to the number of virtual circuits available to that site. The greater the number of virtual circuits which are available the greater the message traffic in the network.

In a network with S sites the number of buffers required at each site is equal to:

$$S \times 2 \times V$$

where V is the maximum number of virtual circuits between each site.

4.5.2.3 Flow Control Data Structures

Figure 4.7 illustrates the data structures used by the flow control layer

the "flow limit" - corresponds directly to the number of virtual circuits available to that site. The greater the number of virtual circuits which are available the greater the message traffic in the network.

In a network with S sites the number of buffers required at each site is equal to:

$$S \times 2 \times V$$

where V is the maximum number of virtual circuits between each site.

4.5.2.3 Flow Control Data Structures

Figure 4.7 illustrates the data structures used by the flow control layer

the "flow limit" - corresponds directly to the number of virtual circuits available to that site. The greater the number of virtual circuits which are available the greater the message traffic in the network.

In a network with S sites the number of buffers required at each site is equal to:

$$S \times 2 \times V$$

where V is the maximum number of virtual circuits between each site.

4.5.2.3 Flow Control Data Structures

Figure 4.7 illustrates the data structures used by the flow control layer

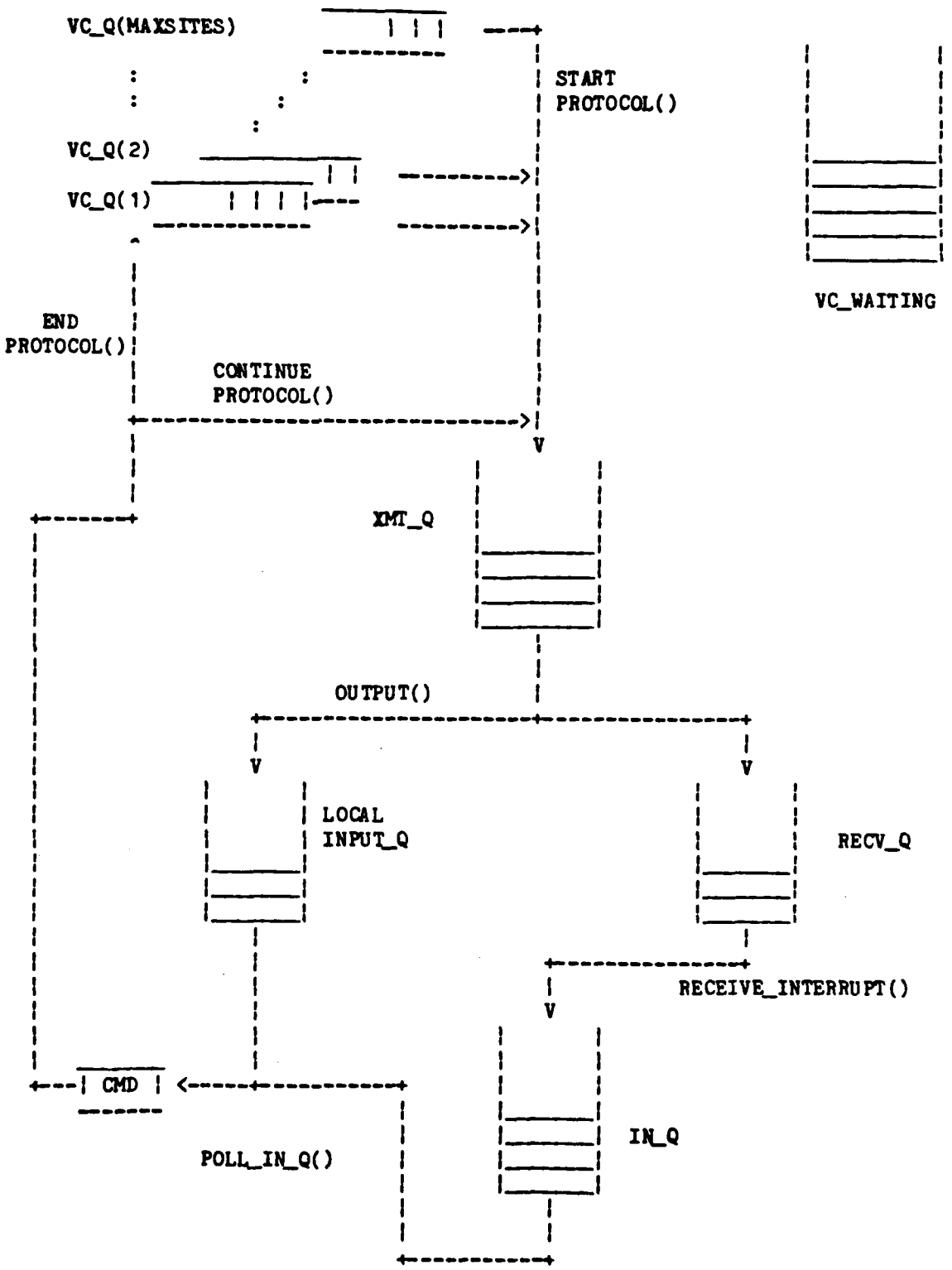


FIGURE 4.7 - FLOW CONTROL DATA ABSTRACTION

VC_Q is a queue of buffer addresses which are available to initiate a virtual circuit. There are FLOW_LIMIT buffers for each site. MAX_SITES is the maximum number of sites in the network. If no buffers are available then a protocol can not be started.

The routine START_PROTOCOL will take a buffer off the VC_Q and place it on the XMT_Q.

XMT_Q is a queue of buffers to be transmitted by the commmunications subsystem. Upon completing a transmission the communications subsystem will place the buffer on the RECV_Q if the message was non local or on the LOCAL_IN_Q if the message was local.

RECV_Q contains buffers which are available for reception of a message by communications subsystem. Upon reception of a message the communications subsystem will place the buffer on the IN_Q.

IN_Q is a queue of messages received by the communications subsystem but not yet serviced by the kernel.

LOCAL_IN_Q is a queue of local messages which have not been serviced by the kernel.

The routine POLL_IN_Q will service the input queues and call the appropriate kernel routines. Depending on which part of the protocol is being executed the buffer will be returned to the VC_Q or the XMT_Q.

CMD points to the buffer which has been taken off the input queues. The kernel will use the information in this buffer to continue a protocol.

VC_WAITING is a linked list of process descriptors which are waiting for a virtual circuit buffer.

Shared Data Structures and Critical Regions

Because interrupt routines can access concurrently (with flow control routines) certain flow control data structures, mutual exclusion must be established on operations which effect these structures. Two techniques were considered.

The first method would turn off processor interrupts upon entering the critical regions. The interrupt procedures would not be able to access the data structures until the interrupts were turned on again. The second method is to use circular buffers to implement the shared date structures. An IN pointer would point to the next place an element could be placed in a queue and an OUT pointer would point to the next element to be taken off the queue. If the concurrent routines access different pointers then mutual exclusion is ensured. In addition, if the capacity of the queue is made one larger than it will ever be, the queue will never be full. Thus when IN is equal to OUT the queue is guaranteed to be empty. The second method was chosen because it was more time efficient.

4.5.2.4 Virtual Circuit Procedures

Five procedures will have access to the flow control data structures:

1. START_PROTOCOL ¦
2. CONT_PROTOCOL > flow control layer
3. END_PROTOCOL ¦

4. XMIT_INTERRRUPT ¦
> network layer
5. RECV_INTERRRUPT ¦

START_PROTOCOL must be called by any kernel routine which may initiate a protocol: CREATE,RUN,SEND,REC,RECF. It will first check to see if a virtual circuit is available. If not, the process descriptor will be queued on VC_WAITING. If a virtual circuit is available a buffer will be removed from VC_Q and the message composed in that buffer. The buffer is then placed on XMT_Q for transmission through the network. The network layer can now be called.

CONT_PROTOCOL is called when the kernel level protocol must be continued. The same virtual circuit will be used to transmit the text of the message. The message is composed in the available buffer and placed on the XMT_Q.

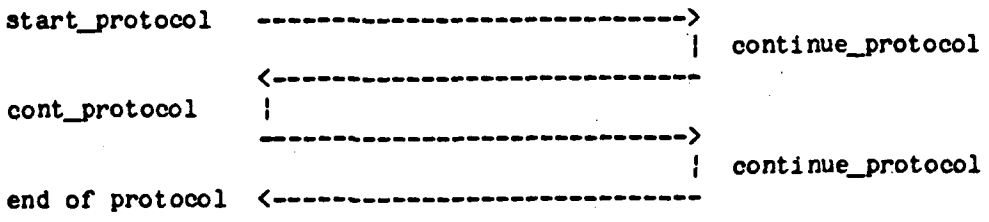
END_PROTOCOL is called when the last part of the protocol is executing. It will first check to see if there are processes waiting to use a virtual circuit. If there are, the waiting process is dequeued from VC_WAITING and assigned the free buffer. The message is then composed and submitted to the network layer. If there are no processes waiting for the virtual circuit the buffer is returned to VC_Q.

XMT_INTERRUPT and REC_INTERRUPT can actually be considered part of the network layer. Upon completion of transmission or reception the I/O devices must be prepared for the next message. A received message

must be placed on the IN_Q. In the case of a transmission the XMT_Q is checked for pending transmissions. If the transmission was local the buffer is placed on the LOCAL_IN_Q otherwise it is placed on the RECV_Q. This ensures that enough buffers will be available to receive the reply. In systems which do not use interrupt I/O (XDCS3) the network layer will queue and dequeue messages in a similar manner.

4.5.2.5 Ramifications of Virtual Circuits on the

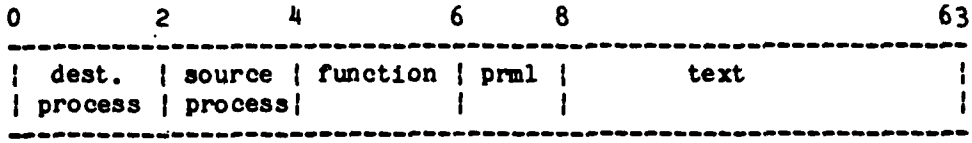
1. The virtual circuit mechanism assumes that each protocol ends at the site where it is initiated, i.e. there are an even number of messages:



In previous versions of the kernel not all of the protocols are implemented this way. SEND for example does not receive a positive acknowledgement after a text message has been transmitted. Another protocol with a similar problem is RUN.

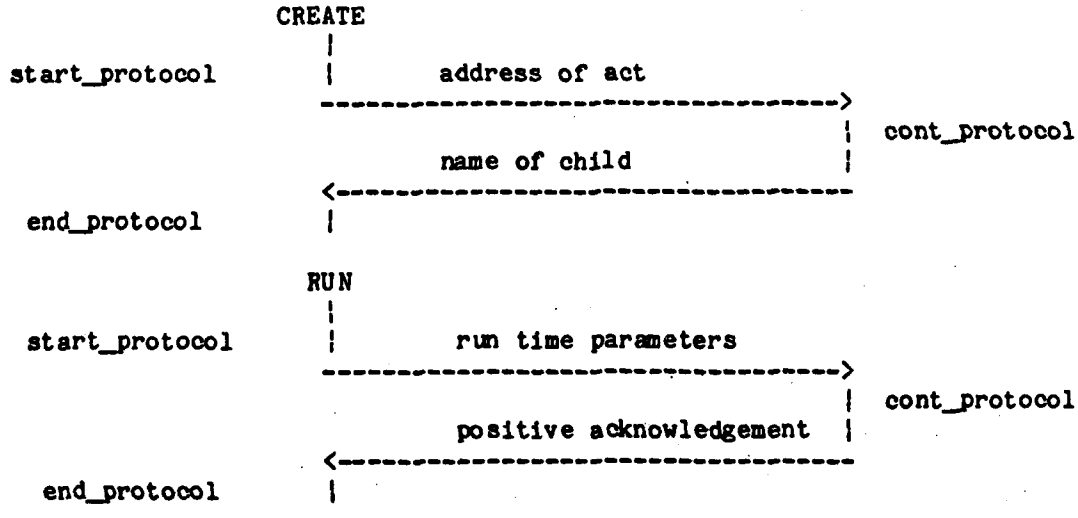
2. Previous versions of the kernel do not use a buffer to hold the message. Messages were transported from process to process. A variable called CMD is a global variable which contains all the information needed to transmit a message of the protocol and respond to one. In particular, it contains the source process i.d., destination process i.d., message type, and the address of additional parameters.

The new version of the kernel will use a buffer abstraction which is similar to CMD except the buffer will hold the text of the message (instead of the address of the text) and the first byte will contain the destination site (needed for the N.I. hardware).

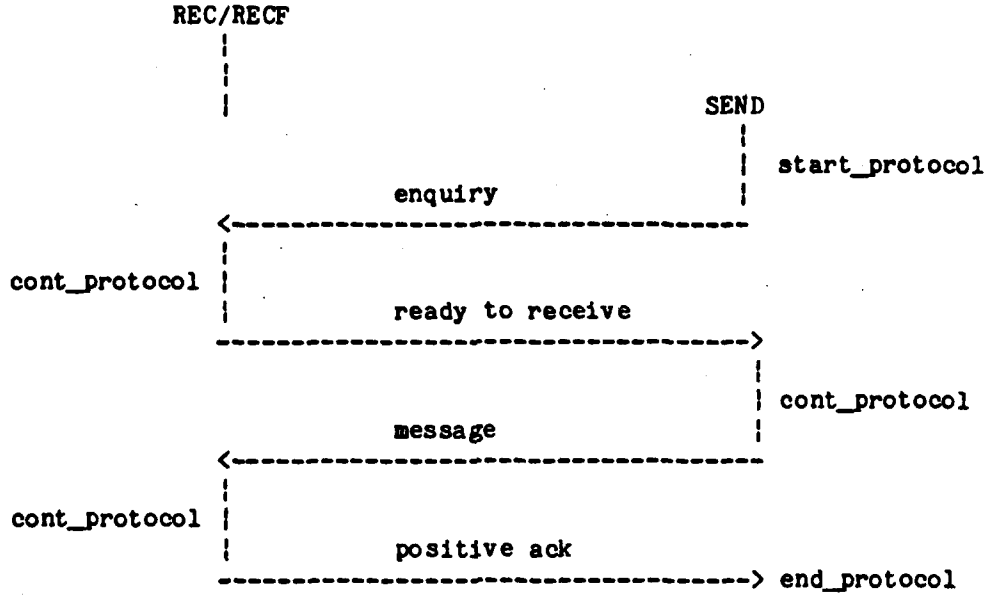


4.5.2.6 Modified Kernel protocols and Flow Control

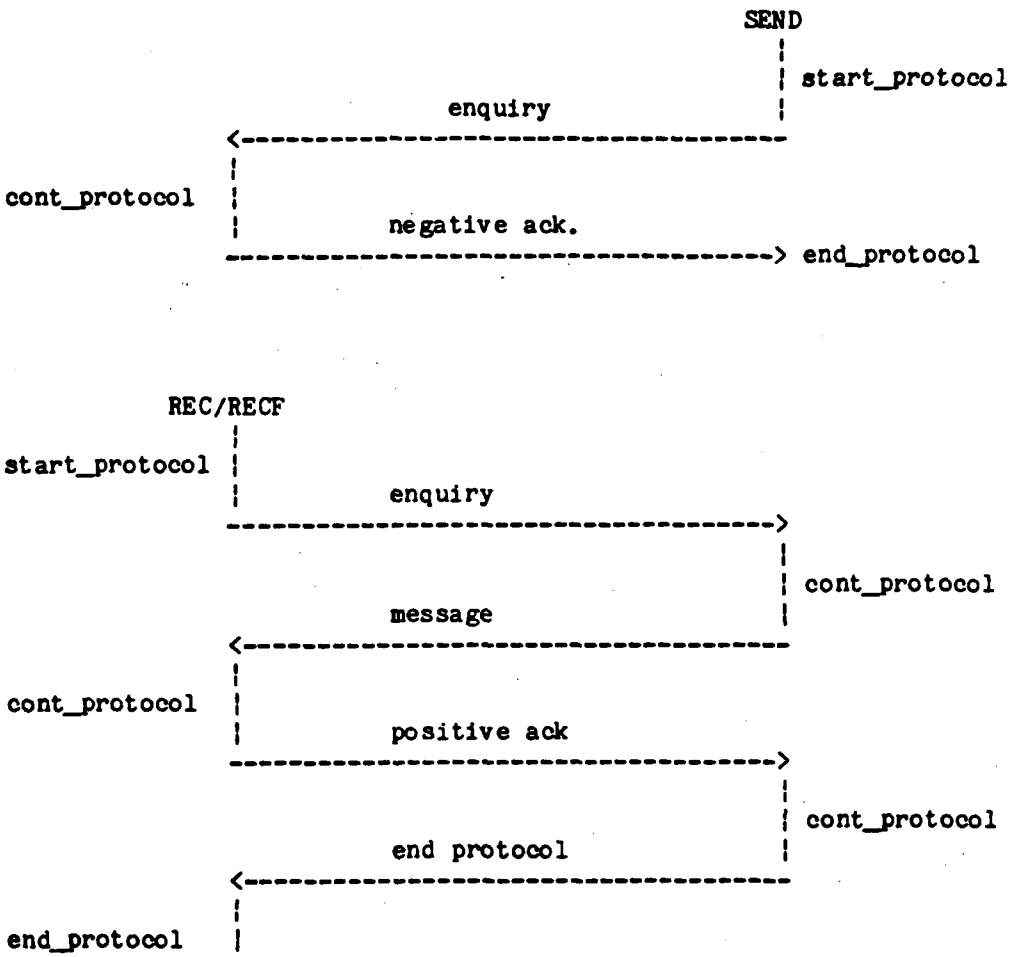
Depending on which part of the kernel protocol which is being executed a different flow control routine will be initiated. The following timing diagrams illustrate when the different flow control routines will be executed:



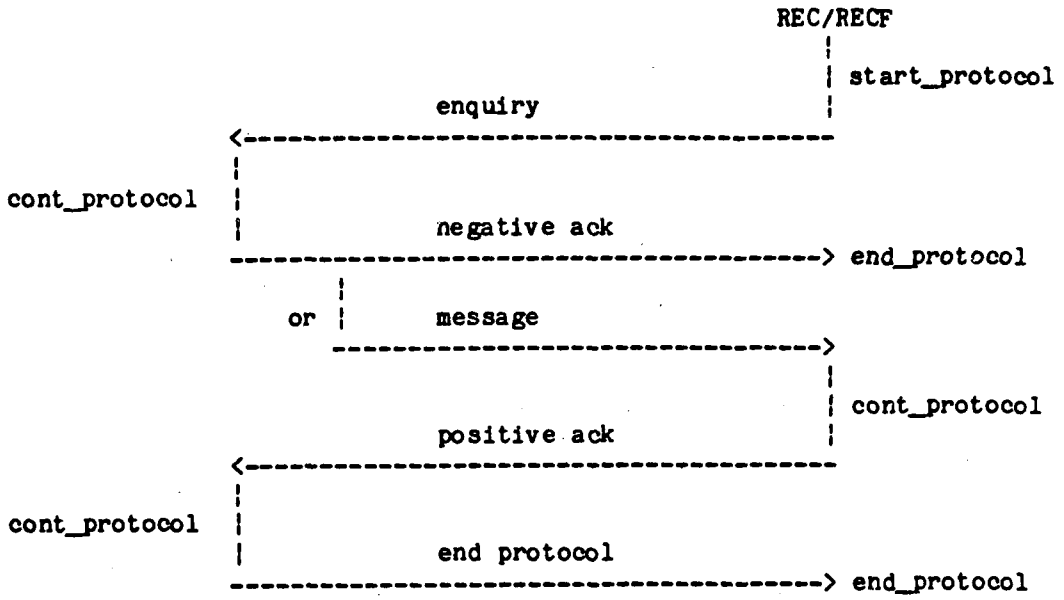
CREATE/RUN



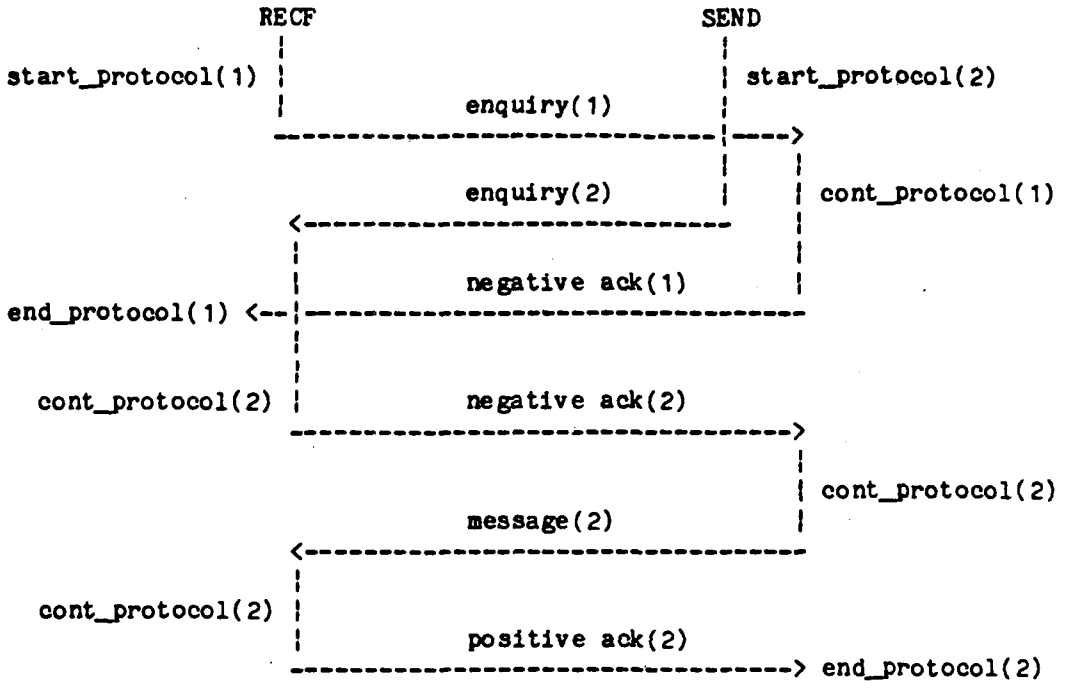
REC first with no pending senders


SEND
start_protocol
enquiry
cont_protocol
negative ack.
end_protocol
REC/RECF
start_protocol
enquiry
cont_protocol
message
cont_protocol
positive ack
cont_protocol
end protocol
end_protocol

SEND first



REC first and pending senders



SEND and RECF concurrently with pending senders

4.5.3 Selection of Message Delivery Technique

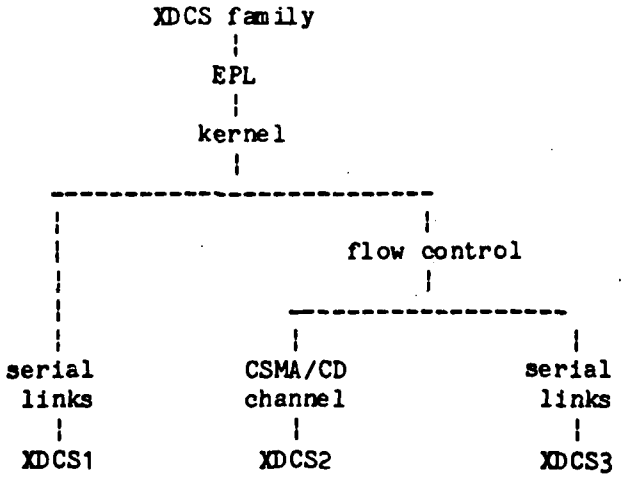
Virtual Circuit flow control was eventually adopted as the method to provide guaranteed message delivery. The reasons for this decision were twofold:

1. Virtual circuit flow control provides the same degree of reliability that was provided in the serial link version (XDCS1).

2. Since both methods (PAR and Virtual Circuits) would work the choice would be based on which method could be implemented more quickly. The sooner the system was implemented the sooner experimentation could begin on it. The judgement was therefore made to implement the Virtual Circuit technique.

4.5.4 Summary - Comparison of Family members.

With the addition of a flow control layer to the XDCS system two new family members have been created. The XDCS family members share software and hardware in the following way:



Each leaf in the tree represents a different family member. As can be seen in the above figure, all family members use the same distributed

language and the same operating system kernel. XDCS1 was the first member of the family to be created and does not use explicit flow control procedures. Flow control is implicit in the link level connection protocol. Contention resolution is also handled in the same contention protocol. Since the serial link network allowed the XDCS1 communications subsystem to examine messages on a byte by byte basis, messages could be routed directly to the destination process. Hence buffering of messages was not required.

Because the CSMA network interfaces were block DMA devices, buffers had to be provided to receive messages. When a message arrives it is DMA'ed into the buffer. Since messages could not be examined on a byte by byte basis they could not be routed directly to the destination process. These latter hardware characteristics required the implementation of the flow control procedures which are documented in the previous sections.

The XDCS2 and XDCS3 systems are identicle except for the network hardware. The primary reason for implementing XDCS3 was to test the flow control concepts with a network that was known to be functioning correctly. The CSMA/CD is a prototype design which had not been thoroughly tested at the time of its acquisition. Also, by using XDCS2 and XDCS3 the effect of two different networks on the performance of the XDCS system could be ascertained (Chapter 5).

Chapter 5

Preliminary Measurements

5.1 Purpose of Measurements

Three systems are now available to obtain measurements on. XDCS1 and XDCS3 differ only in the addition of flow control procedures in XDCS3. They both use the same serial link network and operating system kernel. XDCS2 and XDCS3 differ only with respect to their underlying networks. Therefore the opportunity exists to answer the following questions:

1. How much does the overhead of the flow control procedures effect the performance of the XDCS system?

2. The CSMA/CD network uses a one megabit bus. The serial links provide a bandwidth of 38.2 kilobaud. Given systems with identicle kernels and communications subsystems, does the CSMA/CD based system (XDCS2) provide the same order of magnitude improvement in performance over the serial link system (XDCS3)?

5.2 Benchmarks

The benchmarks used to answer the questions of the previous section are:

1. Communcations Subsystem Bandwidth

This benchmark will determine the bandwidth seen by the communications subsystem. The benchmark consists of a sending process at site 0 and a receiving process at site 1. One thousand messages are transmitted from the sender to the receiver. The average time to transmit a 40 byte message is determined. The experiment is executed on the serial link systems and the CSMA/CD system.

2. Process Bandwidth

This benchmark will determine the bandwidth as seen by EPL processes. A sending process is created at site 0. A receiving process is created at site 1. One thousand messages are transmitted between the processes. The total time to transmit 1000 messages is measured. The total time is divided by 1000 to find the average time to transmit a message. The experiment is performed on all three systems (XDCS1,XDCS2,XDCS3) and is repeated for messages of varying length.

3. Communications Overhead benchmark

This program consists of a pair of processes which transmit 100 messages from the sender to the receiver. By executing this program with 1 and then 2 cpu's, the network communications overhead can be

determined. If both processes of the benchmark are executed by a single cpu, the communcations subsytem uses local memory to transmit messages. If each process is located on a different cpu, the network must be used. When two sites are used, the completion time of the benchmark will be increased by the delays imposed by the network. However, because the computations of the EPL processes may be executed in parallel, there may be some improvement in performance.

The point at which it becomes beneficial to use two processing elements can be determined by varying the amount of computation (percentage of time computing in EPL) between messages in the EPL program.

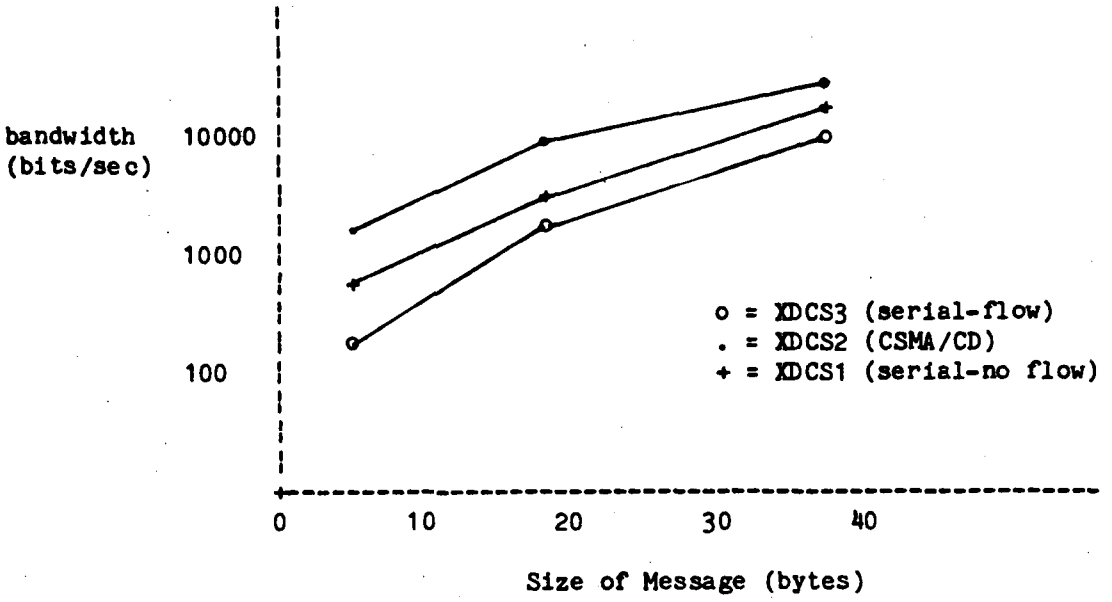
5.3 Results

1. Communications Subsystem Bandwidth:

Serial Link systems: 32,000 bits/sec

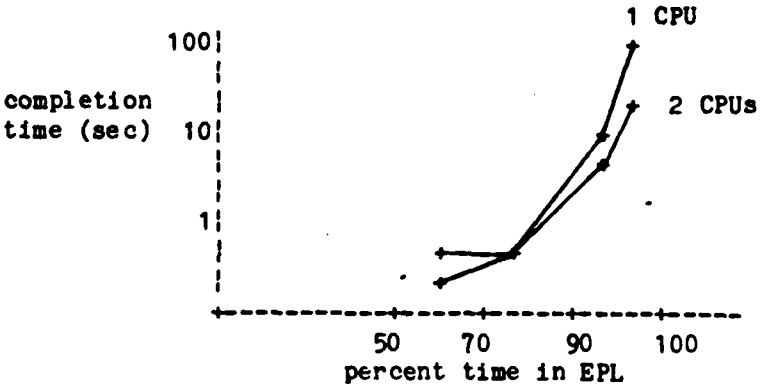
CSMA/CD system: 400,000 bits/sec

2. Process Bandwidth Measurements (2 CPUs)

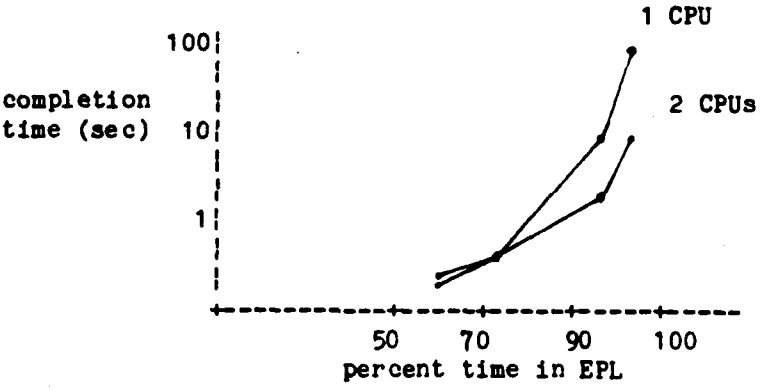


3. Network Overhead Benchmarks

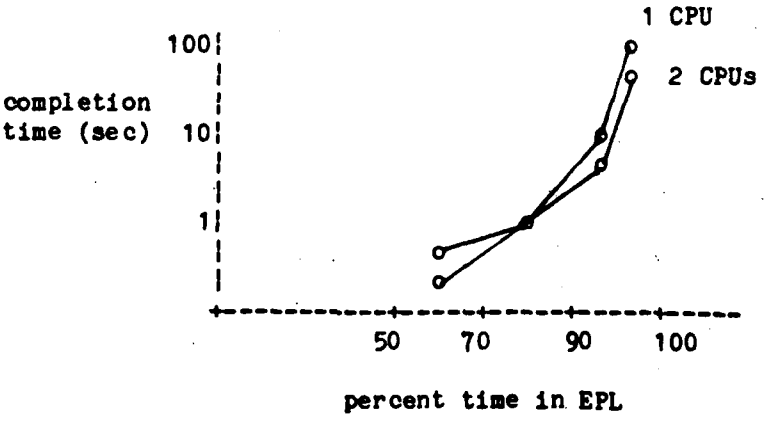
XDCS1 - serial links, no flow control:



XDCS2 - CSMA/CD network:



XDCS3 - serial links, flow control:



5.4 Summary and Conclusions

The communications subsystem bandwidth measurement shows that the CSMA/CD network operates about 12 times as fast as the serial link network. The EPL process bandwidth measurement shows that there is at best a twofold decrease in completion time by using the CSMA/CD channel. This indicates (in the XDCS system) that the communcations software has more impact on performance than does the bandwidth of the underlying network.

The network overhead benchmark shows that when the percentage of time computing in EPL is greater than 70 percent of the total completion time, the communications delays imposed by the network are more than offset by using two processing elements. When the time in EPL is less than 70 percent of the total completion time, it is better to perform the computation on one cpu. This crossover point is not dramatically changed by the system used to perform the measurement.

Chapter 6

Summary

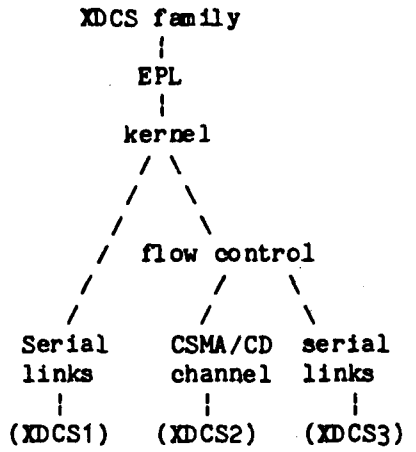
The primary focus of this project was to design a communications subsystem for the XDCS system which was based on a CSMA/CD network. Another goal was to document the XDCS family. Measurements were taken to determine the relative performance of the various family members. A desired side effect of this project was to develop a flexible system in which the various family members could be created by interchanging kernel implementations or communications subsystems.

6.1 The XDCS Family

The characteristics of the XDCS family were described in chapter two. Chapter three presented the implementation of a system which used these characteristics as a guideline for its design (XDCS1). As a result of using the concept of "layers" of software, new family members could be created by changing or adding layers to the XDCS1 system.

In chapter four a new family member (XDCS2) was created by chang-

ing the communications subsystem layer and adding a flow control layer. The following figure illustrates the possible family members which can now be created.



Each path in this tree represents another family member. Each level represents the software or hardware shared by the various family members.

The flow control layer provided the buffers needed for packet switched networks. Systems which use the flow control layer are also flexible enough to work with circuit switched networks such as the serial link network of XDCS1.

Another possibility in flow control systems, is the development of a different kernel level protocol. If messages can be buffered at the receivers site, a sender need not be blocked while waiting for a receiver to become ready to accept the message. The sender can transmit the message and immediately continue running.

The price paid for added flexibility is of course the increased

overhead of the flow control layer. Also, message lengths are limited to a maximum size (64 bytes - which can be increased if needed). This may be restrictive but currently is not a problem.

6.2 Communications Subsystem for the CSMA/CD network

The NI network is a packet switched network which required messages to be buffered at the receiving site. The virtual circuit technique (chapter four) provided a simple way to supply the needed buffer space and implement flow control at the same time.

The disadvantage to the virtual circuit method is that the message transfer error rate (between processes) is directly proportional to the reliability of the underlying network hardware. Experience gained in this experiment showed that prototype hardware may not perform up to its specifications. In situations where the reliability of the network cannot be ascertained, a PAR protocol should be considered.

Another problem is that the flow control algorithm is "keyed" into the kernel protocol. A change in the kernel protocol may adversely effect the operation of the flow control layer.

The virtual channel scheme does work. It provides for buffering of messages to transmitted and received. So far it has been proven successful for both a serial link and a CSMA/CD network. It may therefore be reasonable to assume that the virtual channel scheme will work with other networks as well.

6.3 Performance

The preliminary measurements provided in chapter five indicate that the CSMA/CD based system performs as well or better than the serial link versions. Since increased throughput was not a goal of this project the performance of the CSMA/CD system is at least satisfactory.

The level of performance obtained by the CSMA/CD system was not an obvious result when the added overhead of flow control is considered. The fact that the bandwidth of the CSMA/CD network is 12 times that of the serial link network and that the performance of the systems which use these networks is nearly equal, indicates that the protocols in the XDCS system have a greater impact on performance than does the bandwidth of the network.

6.4 Further Work

The virtual circuit flow control technique provided a simple solution to message buffering in a packet switched network. It provides the degree of reliability specified by the model of a communications subsystem for the XDCS system. If a more robust system is needed, other techniques such as PAR schemes should be considered.

As stated in section 5.1 a kernel protocol can now be considered which is predicated on the fact that buffering is available. A simplified kernel protocol may compensate for the added overhead of the flow control layer.

Further extensions to the XDCS family can still be implemented.

For example a version of EPL with exception handling facilities exists [EEB80]. The kernel should be modified to implement these exception handling primitives. This may eventually lead to the development of communications subsystem software which can detect failures in the network and report the failures to the kernel. The kernel can in turn inform the EPL processes through the exception handling facilities. Users of the system will then be able to investigate fault tolerant algorithms.

Appendix 1 - Details of the EPL/KERNEL interface

This appendix describes the current LSI-11 implementation of the EPL/KERNEL interface.

For all processes EPL expects the first word of the data segment to be initialized to the name of the process. This must be done by the kernel when the data segment is allocated. The EPL keyword SELF refers to this location. When a kernel call is made, the base of the data segment is defined by the value in r0. The value in r1 is a displacement into the data segment where the parameters, needed by the kernel call, can be found. EPL assumes that the kernel will return any results starting at the latter displacement. When the operating system kernel returns control to the EPL process, the process assumes that the success or failure of the operation is indicated by the value in r1. A value of -1 indicates success; a value of 0 indicates failure. Currently the operating system kernel assumes all operations are successful and returns the value -1. The parameters which are passed differ for each kernel call and are discussed subsequently.

1. CREATE - EPL requires that a process be allocated memory at the time of its creation. This allocation takes the form of two data

structures: a process descriptor and a data segment. The size of the process descriptor is constant. The size of the data sement is dependent on the process being created. (For more details of these data structures see the section - Kernel Data Structures - Chapter 2.) Figure A.1 illustrates the data segments before and after the kernel has been requested to create a new process.

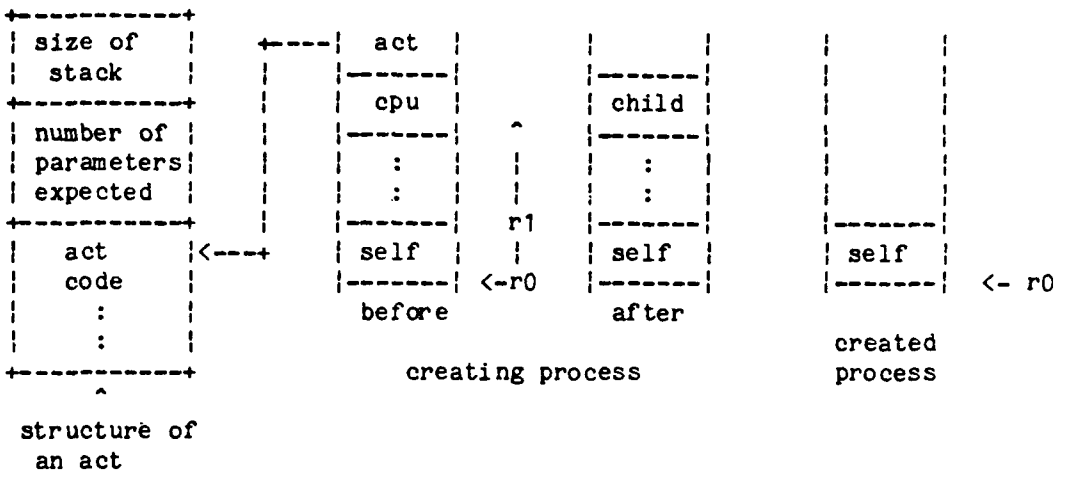


Figure A.1 Creating a process

Parameters passed by EPL are "cpu" and "act". "cpu" specifies the index of the cpu assigned to execute the new process. A negative value indicates that the EPL program has not specified an index. "Child" is the name of the newly created process. The value of "child " is determined by the kernel and returned to EPL. "Act" specifies the address of the first executable statement of code defining the newly created process. It should be noted here that "act" is an address of the start of a program. Therefore, identicle copies of this code must exist at all sites and start at the same location in memory. See appendix 3 for memory layout.

2. RUN - Figure A.2 illustrates the process data segments before and after the operating system kernel has been called to transmit initialization parameter values to a newly created process.

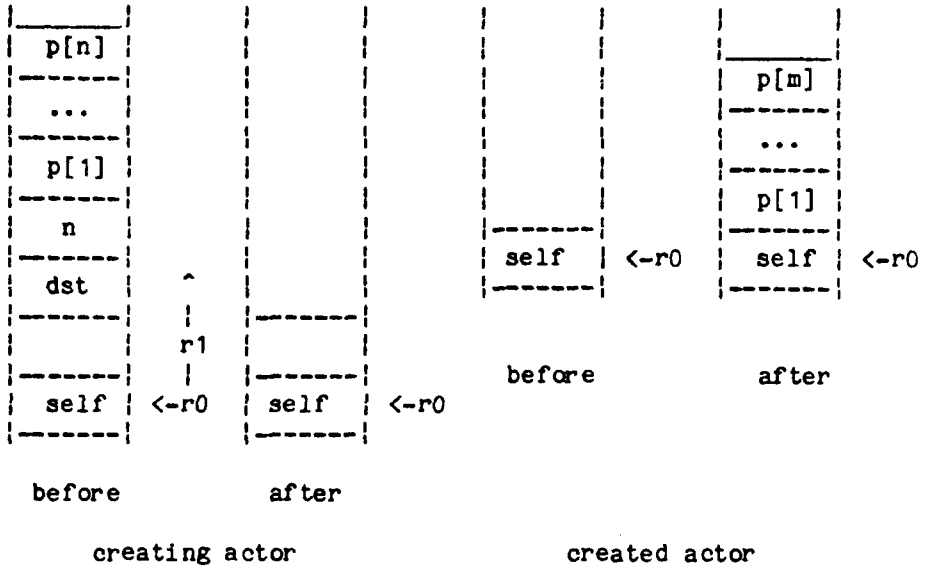


Figure A.2

The parameter "dst" specifies the name of the new process waiting to receive initial parameters. The parameter "n" specifies the number of parmeters to be transmitted. The parameters p[1] ... p[n] are the values to be transmitted. The parameters p[1] ... p[m] are the values that are received to initialize the new process. The value of "m" is that specified in the code segment of the process (see figure A.1).

3. SEND - Figure A.3 illustrates the process data segment of a sending

process before and after a message has been sent by the kernel.

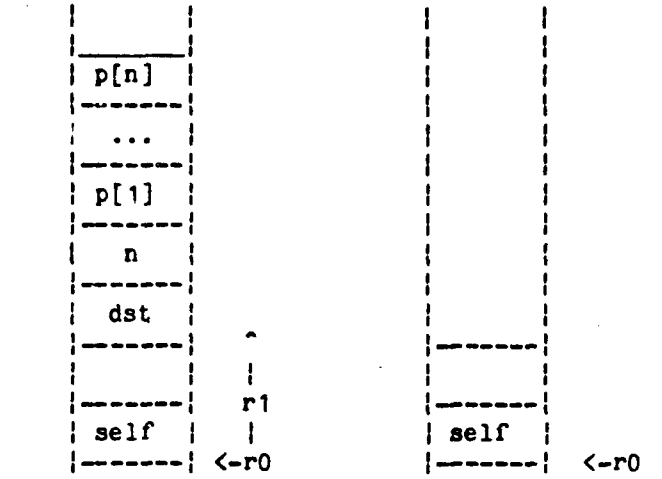


Figure A.3 - Data segment sending process

The parameter "dst" specifies the destination of the message to be transmitted. The parameter "n" specifies the number of parmaters to be transmitted to the destination process. The parameters p[1] ... p[n] are the values to be transmitted to the destination process.

4. REC/RECF - Figure A.4 illustrates the process data segment before

and after a message has been received.

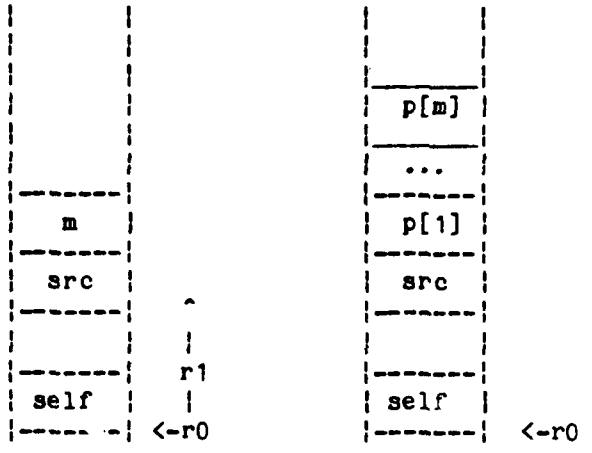


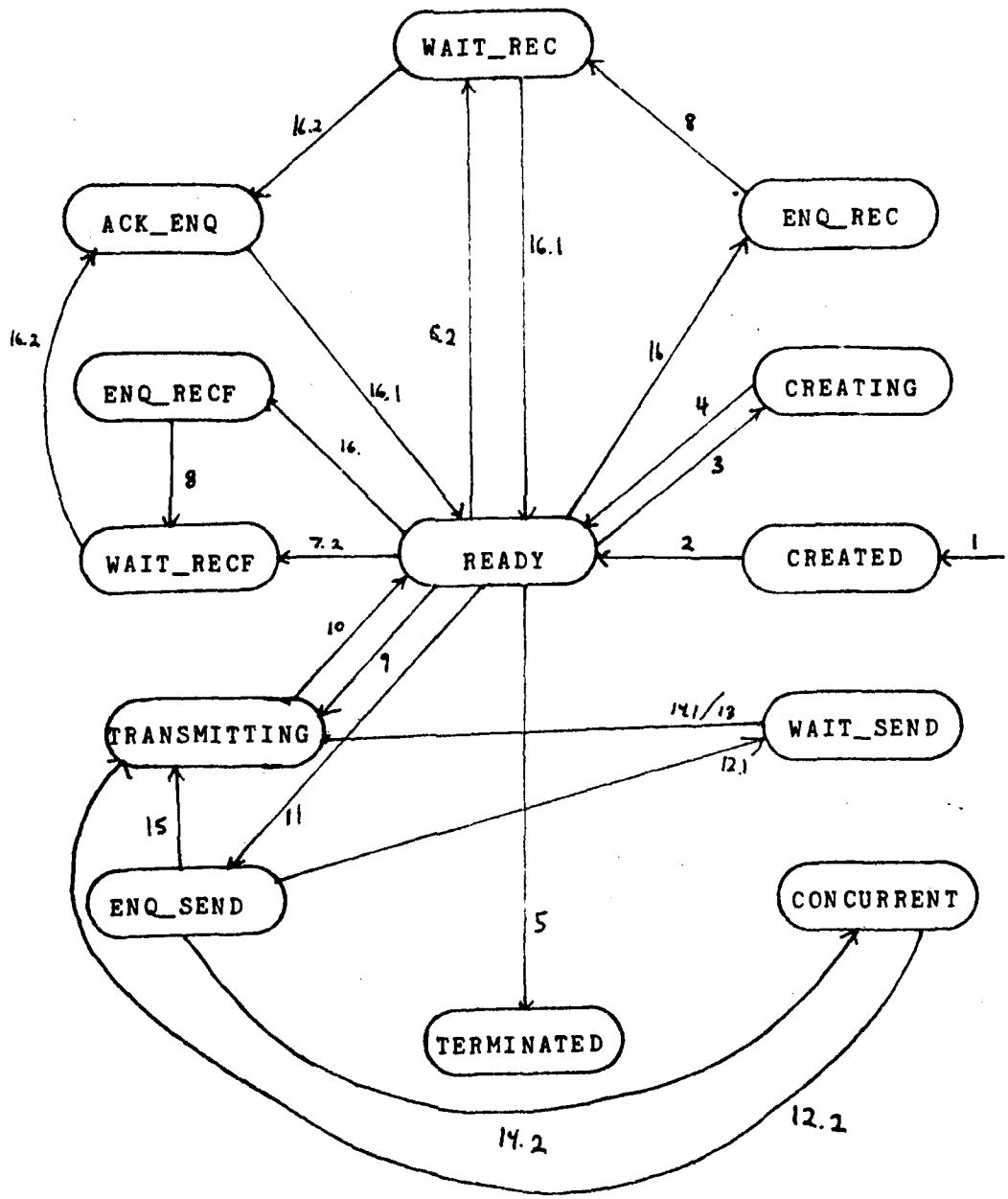
Figure A.3 - Data segment sending process

The parameter "src" specifies the source of the message in the case of the RECF function. Its value is not interpreted by the REC function of the operating system kernel. When the kernel functions of REC and RECF complete, the value "src" contains the name of the process that sent the message. The parameter "m" specifies the number of parameters that the process expects to receive. The parameters p[1] ... p[m] are the values received by the process.

Appendix 2 - State Transitions For XDCS1

State Transition Diagram

State Diagram for the XDCS kernel



State Transition Definitions

ARC KERNEL FUNCTION DESCRIPTION

1. create1 - a new process has been created.

2. run1 - the new process has been given it's runtime parameters and is ready to begin execution.

3. create - a process is creating a new process.

4. create2 - the creating process has received the name of the created process and is ready to execute again.

5. finish - a process has been terminated.

6.1 rec - a process is prepared to receive a message from any sender and there are pending senders. An enquiry will be made to a sender's site.

6.2 rec - a process is prepared to receive a message from any sender and there are no pending senders. No enquiry will be made.

7.1 recf - a process is prepared to receive a message from a specific sender and there are pending senders. An enquiry will be made to the sender's site.

7.2 recf - a process is prepared to receive a message from a specific sender and there are no pending senders. No enquiry will be made.

8. recv2 - a negative acknowledgement has been received from a potential sender.

9. run - a creating process is transmitting the runtime parameters to the created process.

10. done - a process has completed the transmission of data and is ready to begin execution again.

11. send - a process is ready to send a message. It will issue an enquiry to the receiving process.

12.1 send2 - the potential receiver process is not prepared to accept a message.

12.2 send2 - the potential receiver process has concurrently made an enquiry to the sender process. The sender process will transmit the message.

13. send3 - a process is now prepared to receive from any process and has issued an enquiry to a site with pending senders. Upon receipt of the enquiry at the sender's site the kernel will choose a waiting sender and transmit its message.

14.1 send4 - a process is now prepared to receive from a specific process and has issued an enquiry. Upon receipt of the enquiry the sending process will transmit it's message.

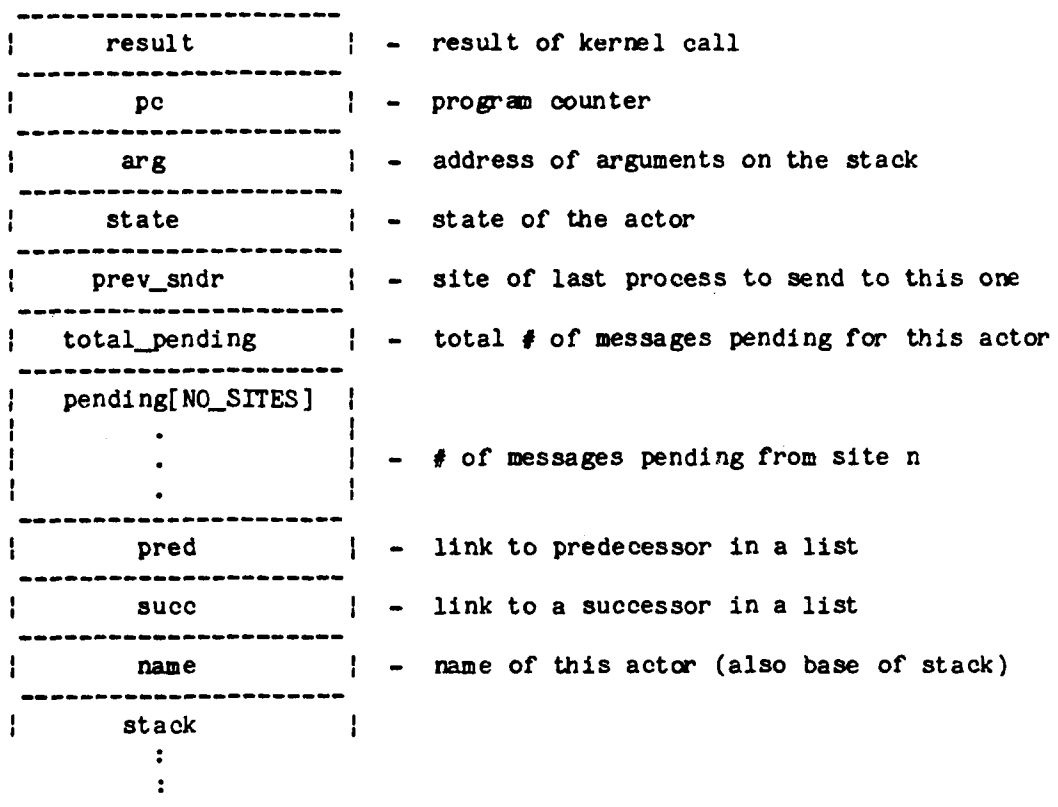
14.2 send4 - a process is now prepared to receive from a specific process and has issued an enquiry. The sending process has concurrently issued an enquiry to the receiving process. The sending process is therefore not prepared to transmit it's message and sends a negative acknowledgement to the receiver process.

15. send1 - in response to an enquiry from a sending process the receiving process has sent a positive acknowledement. The sender transmits it's message.

16.1 recv1 - a receiver process has accepted a message from the sender process and is now prepared to execute again.

16.2 recv3 - a sender has made an enquiry and the receiver has acknowleded that enquiry

Appendix 3 - Process Descriptor for XDCS1



Process Descriptor for XDCS1

Appendix 4 - Format of messages for XDCS1

This appendix describes the format of the messages which are used in the kernel level protocol (Chapter 2). A brief description of the intent of each message is given which is followed by the actual format. The number of bytes is given in parenthesis.

CREATE request: Receipt of a create request message causes the kernel to create a new actor locally. The name of the parent is passed so that the kernel knows where to return the child's name.

function name (2)
name of act (2)
name of parent (2)

CHILD NAME: The child name message is used to return the name of the child to the parent actor.

function name (2)
name of child (2)
name of parent (2)

RUN request: A run request message contains the initialization arguments for a newly created actor. Once the child is initialized it can

be started running.

function name (2)
name of child (2)
name of parent (2)
number of arguments (2)
arguments (variable)

ENQUIRIES: The sending actor's kernel passes an enquiry message to the receiver's kernel to indicate that a sender is prepared to send a message. A receiver may make an enquiry to see if a sender is ready to transmit a message.

function name (2)
name of sender or receiver (2)
name of receiver or sender (2)

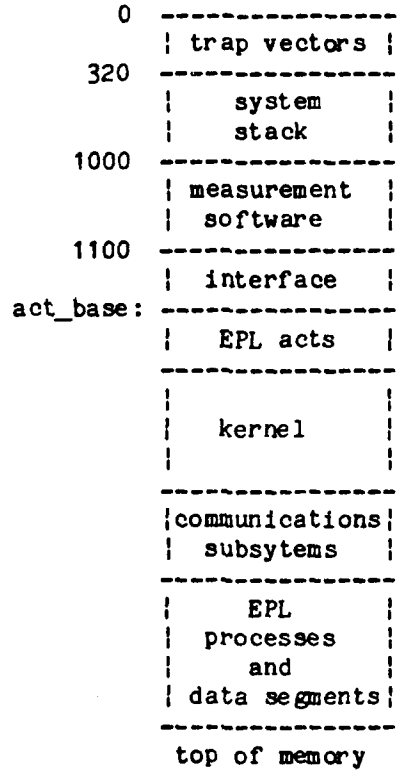
ACKNOWLEDGEMENTS: The acknowledgement message is the response to an enquiry message. It can be either a positive or negative acknowledgement.

function name (2)
name of sender (2)
name of receiver (2)

MESSAGE reception: This message contains the message expected by a receiver and the number of arguments expected.

function name (2)
name of sender (2)
name of receiver (2)
number of arguments (2)
message (variable)

Appendix 5 - Memory Layout and File Description of XDCS1



Interface:

When an EPL process executes a kernel call, a trap is made to the interface. The interface saves the general purpose registers on the system stack. The program counter is placed in the running actors

process descriptor. Upon leaving the interface this information is restored. The interface is responsible for determining which kernel call was made and executing the appropriate code. It will also checks to see if remote sites are trying to send a message. If so, then the appropriate network software is called. The code for starting the kernel begins at location 1100. It initializes the system stack, processor status register, and calls system initialization routines.

Measurements:

This software prints the results of measurements that were installed in XDCS1. Currently these measurements include the percentage of time spent performing each kernel call, time spent in the communications subsytem, the destination of messages, and the number of bytes transmitted to each site. The code to execute the measurement software begins at location 1000.

Act_base:

Act_base marks the beginning of the EPL code.

Bibliography

[BRIN78] P. Brinch Hansen, "Distributed Process: A Concurrent Programming Concept", Communications of the ACM, Nov. 1978, pp. 934-941.

[COHEN80] L. Cohen, S. Fontaine, R. Lenk, and V. Odryna, "Implementation of a Distributed Operating System Using Replicated Kernel Techniques", Advanced Operating Systems Project - CS 358, University of Connecticut, May 1980.

[DEC80] "The Ethernet" - A technical report produced by the Digital Equipment Corporation, Intel Corp., and Xerox Corp., version 1, Sept. 1980.

[DIJKSTRA68] E.W. Dijkstra, "The Structure of the THE - Multiprogramming System", Communications of the ACM, May 1968, pp. 341-346.

[DOD80] Department of Defense, Reference Manual for the Ada Programming Language, July 1980.

[EEB80] E.E. Balkovich, "Decentralized Systems", Technical Report, University of Connecticut, Dec. 1980.

AD-A116 761 CONNECTICUT UNIV STORRS LAB FOR COMPUTER SCIENCE RE--ETC F/G 9/2
A COMMUNICATIONS SUBSYSTEM BASED ON A CSMA/CD CHANNEL.(U)
1981 L S COHEN DAS860-79-C-0117
UNCLASSIFIED TR-CS-82-2 NL

[FLON75] L.Flon, "Program Design with Abstract Data Types", Technical Report, Carnegie Mellon University, June 1975.

[GERL81] M. Gerla, "Routing and Flow Control", Chapter 4 - "Protocols and Techniques for Data Communication Networks", Prentice Hall, 1981.

[GRAY79] J.P. Gray and T.B. McNeill, "SNA multiple system networking", IBM Systems Journal, Vol.18, no.2, pp. 263-297, 1979.

[GUTT80] J. Guttag, "Notes on Type Abstraction (Version 2)", IEEE Transactions on Software Engineering, Jan. 1980, pp. 13-23.

[HAB76] A.N. Habermann, L. Flon, and L. Cooprider, "Modularization and Hierarchy in a Family of Operating Systems", Communcations of the ACM, May 1976, pp. 266-272.

[HOARE78] C.A.R. Hoare, "Communcating Sequential Processes", Communications of the ACM, Aug. 1978, pp. 666-677.

[ISO79] ISO/TC97/SC16, "Reference Model of Open Systems Interconnection", Doc. N227, June 1979.

[KLRK80] L. Kleinrock and M. Gerla, "Flow Control Algorithms: A Comparitive Survey", IEEE Transactions on Communications, April 1980.

[LAM75] S. Lam and L. Kleinrock, "Dynamic Control Schemes for a Packet Switched Multi-access Broadcast Channel", Proceedings of The AFIPS National Computer Conference, pp 143-145, 1975.

[MAY79] M.D. May and R.J.B. Taylor, "The EPL Programming Manual", Distributed Computing Project Report No. 1, Dept. of Computer Science, University of Warwick, Coventry England, 1979

[METCALF76] R.M. Metcalf and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", Communications of the ACM, July 1976.

[MORSE80] J.A. Morse, "NI Controller Functional Specification", Digital Equipment Corporation, Revision 1, Aug. 15, 1980.

[PARNAS72] D.L. Parnas, "On the Criteria To Be Used In Decomposing Systems Into Modules", Communications of the ACM, Dec. 1972, pp. 1053-1058.

[PARN79] D.L. Parnas, "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering, March 1979, pp. 128-137.

[SFONT] S. Fontaine, "A Distributed Computer Operating Systems Kernel", M.S. Thesis, Department of Electrical Engineering and Computer Science, University of Connecticut, Dec. 1980.

[SWARTZ77] M. Schwartz, "Computer Communication Network Design and Analysis", Chapter 13 - "Random Access Techniques", pp. 286-320, Prentice Hall, 1977.

[TOBAG79] F.A. Tobagi and V.B. Hunt, "Performance Analysis of Carrier Sense Multiple Access with Collision Detection", Technical Report - Dept. of Electrical Engineering at Stanford University, June 1979.

[WECKER80] S.WECKER, "DNA: The Digital Network Architecture", IEEE Transactions on Communications, April 1980.

[ZIM80] H. Zimmerman, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection", IEEE Transactions on Communications, April 1980.


ILMED
8